STUDY OF NON-COMPARISON SORTING ALGORITHMS

By

ZHIMIN MA

Bachelor of Science

Beijing University of Aeronautics & Astronauntics

Beijing, P.R.China

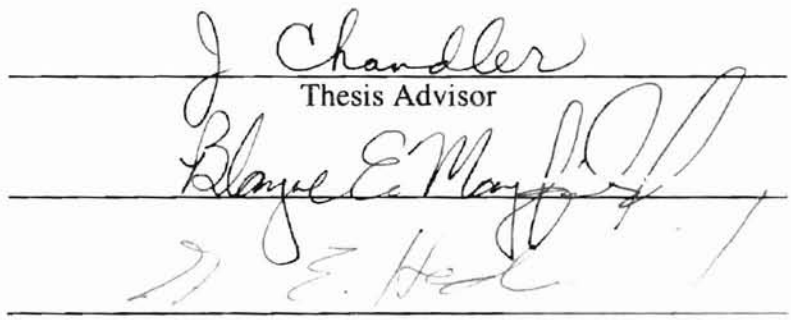
1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2000

STUDY OF NON-COMPARISON

SORTING ALGORITHMS

Thesis Approved:



Thesis Advisor

Dean of the Graduate College

PREFACE

The primary purpose of this project is to compare two new non-comparison sorting algorithms: Groupsort and Flashsort1. To simplify discussion, we focus on the performance of these algorithms with integer data. First, we test Groupsort and Flashsort1 against bucket sort using uniformly distributed integer values with a number of different additional storage spaces and give run time performance curves. Then, we test Groupsort and Flashsort1 against Quicksort using different distributed input data and give run time perfomance curves. Through the analysis of impact of run time, additional storage space, and data distribution, an optimal method is given to make each algorithm perform well.

I sincerely thank my M.S. Committee — Drs. J. P. Chandler, G. E. Hedrick, and B. E. Mayfield --- for guidance and support in the completion of this research.

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my advisor, Dr. J. P. Chandler, for his intelligent supervision, constructive guidance, and inspiration. My sincere appreciation extends to my other committee members Dr. G. E. Hedrick, and Dr. B. E. Mayfield, whose guidance, assistance, and encouragement were also invaluable.

I would like to give my special appreciation to my parents Mr. Qinggui Ma and Ms. Chenggui Zhang for their love and understanding throughout my graduate study. No words can express my affection to my husband, Guang Liu, for his endless love, encouragement and support, and to my daughter, Elizabeth Fangxi Liu, for her enticing smile.

Finally, I would like to thank the Department of Computer Science for support during these three years of study.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 4

2.1 Sorting and its Classification 4
2.2 Comparison-Based Sorting Algorithms 5
2.3 Non-Comparison Sorting Algorithms 9

III. GROUPSORT 13

3.1 Sorting Algorithm 13
3.2 Flowchart for Groupsort 16
3.3 Theoretical Running Time Analysis of Groupsort 16
3.4 The C Programming Code for Groupsort 18

IV. FLASHSORT1 19

4.1 Sorting Algorithm 19
4.2 Flowchart for Flashsort1 20
4.3 Theoretical Running Time Analysis of Flashsort1 23
4.4 The C Programming Code for Flashsort1 23

V. COMPUTATIONAL RESULTS 24

5.1 Run Time Results for Different Additional Storage Space 24
5.2 Run Time Results for Different Data Distribution 24

VI. COMPARISON OF GROUPSORT AND FLASHSORT1 28

6.1 Running Time and Their Impact 28
6.2 Additional Storage Space and Their Impact 33
6.3 Data Distribution and Their Impact 40

VII. CONCLUSIONS AND IMPROVEMENTS 47

7.1 Conclusions 47
7.2 Improvements 47

Chapter Page

BIBLIOGRAPHY 48

APPENDIXIES 51

APPENDIX A---C PROGRAMMING CODE FOR GROUPSORT 52
APPENDIX B---C PROGRAMMING CODE FOR FLASHSORT1 60
APPENDIX C---C PROGRAMMING CODE FOR BUCKET SORT 65
APPENDIX D---C PROGRAMMING CODE FOR QUICKSORT 69
APPENDIX E---C PROGRAMMING CODE FOR GENERATING UNIFORMLY DISTRIBUTED INPUT ARRAY 72
APPENDIX F---C PROGRAMMING CODE FOR GENERATING TRUNCATED NORMAL DISTRIBUTED INPUT ARRAY 74

LIST OF TABLES

Table Page

1. Summary of seven general comparison-based sorting algorithms 8

2. Relative running times under uniformly distributed integer array 26

3. Relative running times under truncated normal distributed integer array 27

4. Running speed comparison with additional storage space $0.1n$ 40

LIST OF FIGURES

Figure Page

1. An example of using Groupsort 15
2. Flowchart for an efficient implementation of Groupsort 17
3. An example of using Flashsort1 21
4. Flowchart for an implementation of Flashsort1 22
5. Relative running times with list size for additional storage space $0.05n$ 29
6. Relative running times with list size for additional storage space $0.1n$ 30
7. Relative running times with list size for additional storage space $0.2n$ 31
8. Relative running times with list size for additional storage space $0.3n$ 32
9. Relative running times with additional storage space for $n = 2000$ 34
10. Relative running times with additional storage space for $n = 5000$ 35
11. Relative running times with additional storage space for $n = 10000$ 36
12. Relative running times with additional storage space for $n = 15000$ 37
13. Relative running times with additional storage space for $n = 20000$ 38
14. Relative running times with additional storage space for $n = 25000$ 39
15. Relative running times with different standard deviations for $n = 2000$ 41
16. Relative running times with different standard deviations for $n = 5000$ 42
17. Relative running times with different standard deviations for $n = 10000$ 43
18. Relative running times with different standard deviations for $n = 15000$ 44

Figure Page

19. Relative running times with different standard deviations for $n = 20000$ 45

20. Relative running times with different standard deviations for $n = 25000$ 46

CHAPTER I

INTRODUCTION

Sorting is one of the fundamental problems in computer science because of the following reasons. First, it is the basis of many other algorithms such as searching, pattern matching, etc. Many applications have been found in database systems, data statistics and processing, data communications and pattern matching [19]. Second, it plays an important role in the learning of algorithm design and analysis, data structure and programming. Sorting algorithms make a valuable case study in efficient coding. Through studying various sorting algorithms, we can find many important principles of data structure manipulation, and discover the evolution of various sorting algorithms. By examining the evolution, we can learn the general ideas involved in the analysis of algorithms — the ideas used to determine performance characteristics of algorithms so that an intelligent choice can be made between competing methods, and more further we can learn a good deal about strategies that help us design good algorithms. Finally, and especially, it is a challenging problem that has been studied extensively [13, 14, 15, 19, 24, 25]. The performance of these algorithms has been improved dramatically [12, 20, 25, 26]. The proven lower bound of complexity has been reached [1, 7, 19, 24, 31]. It has shown that various sorting algorithms have their own characteristics.

On comparing the various methods that have been achieved, sorting algorithms can be divided generally into comparison sorting algorithms and non-comparison sorting algorithms. Regardless of the model used, we need to consider the following general aspects in designing an algorithm for sorting n records:

1. Computational efficiency in both the theoretical and practical sense, including analyzing and determining the running time.
2. The amount of additional storage space used by a sorting algorithm and the ability to control the additional storage space used.
3. Implementation and use including how difficult the programming is and how data distribution affects the performance.

In light of these criteria, no sorting algorithm is perfect, not even Quicksort, which is considered to be the best available general-purpose, comparison-based sorting algorithm. As presented first by Hoare [14] and subsequently in Sedgewick [25] for variations, the average running time of Quicksort is $O(n\log_2 n)$, but its worst-case behavior is $O(n^2)$. Conventional comparison-based sorting algorithms and their average-case time complexities include selection sort ($O(n^2)$), insertion sort ($O(n^2)$), bubble sort ($O(n^2)$), Shellsort (two conjectures: $O(n(\log_2 n)^2)$ and $O(n^{1.25})$), Quicksort ($O(n\log_2 n)$), Heapsort ($O(n\log_2 n)$), Mergesort ($O(n\log_2 n)$), etc., as described by Knuth [19]. By using a decision-tree model, a lower bound of $\Omega(n\log_2 n)$ on the worst-case running time of any comparison sort on n inputs has been proved [1, 7, 19, 24, 31]. In order to avoid this performance limitation of comparison sorts, various non-comparison methods [19], such as address-calculation sorts, distribution counting sorts, radix sorts, counting sort and bucket sorts, have been designed which use the values of the elements being sorted to increase efficiency to $O(n)$, but do so at the expense of additional storage space and a lack of generality.

The sorting algorithms themselves are not difficult to understand, but a comparison of the relative merits of the many algorithms does require some effort. The

goal of this study is to compare two new non-comparison sorting algorithms, Groupsort [6] and Flashsort1 [21]. This study focus mainly on the performance of these algorithms with positive integers. I use the C programming language to implement the algorithms, and give computational results for running time including running time for various sorting array sizes, running time for various additional storage spaces and running time for various data distributions. Based on these results, this study will analyze the time complexity, the additional storage space used and the affection of the data distribution, discuss the advantages and disadvantages of each algorithm, and give some conclusions and suggestions of improvements.

CHAPTER II

LITERATURE REVIEW

2.1 Sorting and its Classification

Sorting is used to put items in order. Consider the problem of sorting *files* of *records* containing *keys* which are used to control the sort. A formal description of sorting can be defined based on the concept of partial order. As preparation the definition of partial order is:

Definition 1. Suppose R is a relation on a set S. For a, b, c in S, if R is:

a) Reflexive: aRa for every a in S;

b) Transitive: $aRb \wedge bRc \Rightarrow aRc$; and

c) Antisymmetric: $aRb \wedge bRa \Rightarrow a = b$,

then R is a partial order on the set S.

Sorting generally is defined as arranging a list of data by their keys or themselves into a partial order R, where R implies $\leq$ particularly. The definition is as follows.

Definition 2. For n items $a_1, a_2, \ldots, a_n$ in a set S with each item a_i consisting of a key K_i and some information associated with that key, sorting is an arrangement of the items in order to obtain a partial order K_iRK_{i+1} for $\forall$ i, $1 \leq i < n$.

Generally, R is defined as $\leq$ in sorting, so that the partial order is

$$K_1 \leq K_2 \leq \ldots \leq K_i \leq \ldots \leq K_n$$

On comparing the various methods that have been achieved, sorting algorithms can be classified into two categories: (1) the atomic comparison model, which assumes that the sorted order is based on comparison between the input items and (2) the non-comparison model in which the value of an individual item is used to estimate its location in the final sorted list. Many examples of both types of sorting algorithms are found in Knuth [19] and Cormen et al. [7].

2.2 Comparison-Based Sorting Algorithms

2.2.1 Comparison-Based Sorting Algorithms

First, let us start from some conventional comparison-based sorting algorithms: bubble sort, straight selection sort, straight insertion sort, Shellsort, Quicksort, Heapsort, and Mergesort.

1. Bubble sort [17, 18, 28, 29]:

Bubble sort is one of the simplest interchange sorts. The bubble sort algorithm sorts an array by interchanging adjacent items that are in the wrong order. The algorithm makes repeated passes through the array probing all adjacent pairs until the file is completely in order.

The bubble sort is a simple sorting algorithm, but it is inefficient. Its running time is $O(n^2)$, unacceptable even for medium-sized files. One advantage of the bubble sort is that it is stable [11]: items with equal keys remain in the same relative order after the sort as before.

2. Straight selection sort [19]:

The straight selection sort is one of the simplest sorting algorithms that works as follows: first find the smallest item in the array and exchange it with the item in the first position, then find the second smallest item and exchange it with the item in the second position, continuing in this way until the entire array is sorted.

Its running time is $O(n^2)$. It works very well for very small files.

3. Straight insertion sort [19]:

An algorithm almost as simple as straight selection sort but perhaps more flexible is straight insertion sort. For sorting a row vector from left to right, the item being considered is inserted merely by moving larger items one position to the right, then inserting the item into the vacated position.

Its running time is $O(n^2)$. For this reason, the use of the algorithm is justifiable only for sorting very small files. Straight insertion sort is slower than straight selection sort unless the data already have considerable order. One advantage of the straight insertion sort is that it is stable, as is bubble sort.

4. Shellsort [3, 4, 22, 24, 30]:

Shellsort is a simple extension of straight insertion sort which allows exchanges of items that are far apart. Several passes are made through the list of items. After each pass, the items are more nearly sorted. The last pass is just straight insertion sort, but no item has to move very far.

The average running time of Shellsort is not exactly known. Two conjectures are $O(n(\log_2 n)^2)$ and $O(n^{1.25})$. Shellsort is not a stable sorting algorithm since equal keys may not have their original relative ordering after a pass. Shellsort seems a very attractive

algorithm for internal sorting because it has acceptable running time even for moderately large files.

5. Quicksort [13, 14, 15, 16, 27]:

Quicksort is a "divide and conquer" method for sorting. To begin each iteration, a key value *v* is selected from the file as a *pivot* item. The file is then split into two subfiles, those items with keys smaller than the selected one and those items whose keys are larger. In this way, the selected item is placed in its proper final location between the two resulting subfiles. This procedure is repeated recursively on the two subfiles and so on.

Quicksort is considered to be the best available general-purpose, comparison-based sorting algorithm; although its worst case running time is $O(n^2)$, its average performance is excellent ($O(n\log_2 n)$).

6. Heapsort [5, 32]:

Heapsort is a sorting algorithm that sorts by building a priority queue. The idea is simply to build a heap containing the items to be sorted and then to remove them all, in order, using the basic operations on heaps.

Heapsort is guaranteed to execute in $O(n\log_2 n)$ time even in the worst case. With R. W. Floyd's improvement [9], Heapsort is only about 14% slower than Quicksort on the average and much faster in the worst case [19]. Heapsort does not benefit from a sorted array, nor is its efficiency significantly affected by any initial ordering. This algorithm does not use any extra storage. All of these advantages noted indicate that Heapsort is an excellent choice for an internal sorting algorithm.

7. Mergesort [2, 5]:

Mergesort is a natural way of sorting lists by repeatedly merging sublists. By counting the total number of items in the list, each merging step can be as balanced as possible. At the deepest level of the recursion, single item lists are merged together to form two-item lists and so on.

Mergesort is guaranteed to execute in $O(n\log_2 n)$ even in the worst case. It can take advantage of partially ordered lists. But Mergesort uses extra storage: either an auxiliary array or the pointers that are associated with the list. In view of the above, Mergesort is one of the best alternatives for sorting a list, and is best for external sorting.

2.2.2 Summary of Comparison-Based Sorting Algorithms

Table 1 gives a summary of the analysis of the seven general comparison-based sorting algorithms.

Table 1. Summary of seven general comparison-based sorting algorithms

Algorithm	Bubble sort	Straight selection	Straight insertion	Shellsort	Quicksort	Heapsort	Mergesort
Worst case run time	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^{1.5})$*	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$
Average case run time	$O(n^2)$	$O(n^2)$	$O(n^2)$	unknown**	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
Storage requirement	in place	in place	in place	in place***	extra space $O(\log_2 n)$	in place	extra space $O(n)$
Stable	Yes	no	yes	no	no	no	yes

* V. R. Pratt's version of Shellsort runs in $O(n(\log_2 n)^2)$ in the worst case, but its large coefficient of proportinality makes it impratical [4, 22].

** Two conjectures are: $O(n(\log_2 n)^2)$ and $O(n^{1.25})$ [24].

*** In-place: a sorting algorithm sorts in place if only a constant number of elements of the input array are ever stored outside the array [7].

2.2.3 Analysis of Lower Bound on the Running Time

Comparison-based sorting algorithms assume that the sorted order is based on comparison between the input items, the pair of keys compared at any moment of time depends only on the output of previous comparisons and nothing else. We can draw a binary tree in which each internal node is annotated by $a_i : a_j$ for some i and j in the range $1 \leq i, j \leq n$, where n is the number of items in the input sequence. Each leaf is annotated by a permutation. This is called a *decision tree*. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. The length of the longest path from the root of a decision tree to any of its leaves represents the worse case number of comparisons the sorting algorithm performs. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison-based sorting algorithm. By using this decision-tree model, a lower bound of $\Omega(n\log_2 n)$ on the worse case running time of any comparison-based sorting algorithm on n items has been proved [1, 7, 19, 24, 31].

2.3 Non-Comparison Sorting Algorithms

In order to break the limitation of comparison-based sorting algorithms and achieve faster algorithms, various non-comparison algorithms have been designed that use the values being sorted to increase efficiency to O(n), but do so at the expense of additional storage space and a lack of generality.

2.3.1 Basic Non-Comparison Sorting Algorithms

1. Counting Sort [7]:

Counting sort assumes that each of the n records that will be sorted is a distinct integer whose key is between 1 and k. The basic idea of counting sort is to count the number of keys with each value, then use the counts to move the records into position in the output array. This scheme must be modified slightly to handle the situation in which several records have the same value.

The overall running time is O(k + n). In practice, we usually use counting sort when k = O(n), in which case the running time is O(n). An important property of counting sort is that it is stable.

2. Radix Sort [7]:

For many applications, the keys used to define the order of the records in the final sorted list can be thought of as numbers from some range. Sorting methods that take advantage of the digital properties of these numbers are called radix sorts. Consider sorting n records in which each key has d digits and each digit is in the range 1 to k, and k is not too large. Radix sort sorts on the least significant digit first, then the second until all digits have been treated. For each digit, counting sort is the obvious choice.

The total running time of radix sort is $\Theta(dn + kd)$. When d is constant and k = O(n), radix sort runs in linear time. But radix sort that uses counting sort as the intermediate stable sort does not sort in place.

3. Bucket Sort [7]:

Bucket sort assumes that the values being sorted are uniformly distributed throughout the range. The idea of bucket sort is to divide the records into K equal-sized subranges, or *buckets*, and then distribute the n records into the K buckets. To produce

the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the records in each.

The average running time of bucket sort is linear under the assumption that the values being sorted are uniformly distributed. If this assumption is violated, the worst case for bucket sort is all data at one bucket except one at another bucket. The worst case time can be $O(n\log_2 n)$. The disadvantage of this approach include: the use of the linked list data structures and the need for additional storage for the n linked list records in the buckets together with K pointers pointed to the head of each list.

2.3.2 Two New Non-Comparison Sorting Algorithms

Dozens of non-comparison sorting algorithms have been invented based on the above basic non-comparison sorting algorithms. They focus on retaining the linear running time and reducing the additional storage space. An early approach that developed by Gamson and Picard [10] used the additional storage space 3n. A subsequent improvement by Ducoin reduced the additional storage requirements to 1.5n [8]. Two new non-comparison sorting algorithms have been proposed recently.

1. Groupsort:

Recently, Burnetas, Solow, and Agarwal [6] proposed a specific implementation of a bucket sort called Groupsort which is a modification of the one in Ducoin [8] Groupsort retains the linear average running time for uniformly distributed numbers and the ability to sort without using linked lists. It is accomplished by partitioning the array X into K subarrays corresponding to the K buckets, and then moving the numbers from their original locations to the correct subarray, based on an interpolation function identical to the one used in interpolation search [19]. The main improvements of

Groupsort compared with the one in Ducoin are: (1) the reduction in additional storage from 1.5n to ρn, where ρ is any user-chosen value between 0 and 1, and (2) the use of Groupsort or Quicksort to sort the numbers in each bucket, thus improving on the use of insertion sort when the number of items in a bucket is large.

If the uniformly distributed assumption is violated, the worst case for Groupsort is that all data at one bucket except one at another bucket. The worst case time is $O(n^2)$.

2. Flashsort1:

Another interesting sorting algorithm, called Flashsort1, was presented by Neubert [19] in 1998. The algorithm consists of three logical blocks: classification, permutation, and straight insertion. Classification determines the size of each class of items. Permutation does long-range reordering to collect items of each class together. And a straight insertion does the final short-range ordering. It sorts in linear time and requires less than 0.1n auxiliary storage to sort n items, for uniformly distributed keys.

If the uniformly distributed assumption is violated, the worst case for Flashsort1 is that all data at one class except one at another class. The worst case time is $O(n^2)$.

From the above review, we find that every sorting algorithm has its own characteristics. An interesting topic is the comparison of Groupsort and Flashsort1 which all have linear running time performance and use only a little bit additional storage to sort n items for uniformly distributed data. In the following chapters, we will implement these two algorithms using the C programming language and analyze their performance for positive integers to simplify the discussion.

CHAPTER III

GROUPSORT

3.1 Sorting Algorithm

The following description of the Groupsort algorithm is quoted from [6].

"Groupsort breaks the range of values, say u to v, into K subranges of equal length, also referred to as *intervals*, numbered 0 through $K - 1$. Hereafter, the set of elements of the array being sorted that fall within each subrange is referred to as a *group*.

By using one additional pass through the array to count the number of elements that fall in each group, we can obtain an additional array *Count* of K elements that stores the number of elements in each group. Knowing the number of elements in each group allows one to partition the original array x into K subarrays (of varying lengths) that will eventually hold the numbers in each of the K groups.

Once the elements of x are moved to their proper group, the numbers in each group must be sorted to obtain the final list. Because in Groupsort the numbers in each group are stored in consecutive positions of the array, it is possible to use Quicksort to do so rather than the less efficient insertion sort as other bucket sorts do."

Here is an example of using Groupsort. Suppose there are 12 keys, stored in *x[1]*, ..., *x[12]*, whose values range from $u = 1$ to $v = 200$, as shown in Figure 1a. Suppose that the group number $K = 4$. Then the 4 groups are:

Group	Range of values
0	1 — 50
1	51 — 100
2	101 — 150
3	151 — 200

Through passing the array in Figure 1a, we can determine the number of elements (3, 3, 2, and 4, respectively) falling in each group. So, the original array x can be partitioned into the 4 subarrays shown in Figure 1b, in which the pointer (arrow) numbered k indicates the starting position for the numbers that will eventually be put in group K.

Having determined the starting location of each subarray, for each $i = 1, \ldots, n$, $x[i]$ is exchanged with the element in the next available position of the subarray in x to which $x[i]$ belongs. In this example, the first key, whose value is 75, belongs to group 1 and is therefore exchanged with the next available element in subarray 1 of the array x, namely, $x[4] = 98$, as seen in Figure 1c. The value 98 belongs to group 1 and is therefore exchanged with the next available element in subarray 1 of the array x, namely, $x[5] = 1$, as seen in Figure 1d. This exchange process eventually results in all numbers being placed in their appropriate group, as illustrated for this example in Figure 1e.

Finally, Quicksort is used to sort each group to obtain the final list, as shown in Figure 1f.

Values	75	32	200	98	1	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12

a. The original array to be sorted

Values	75	32	200	98	1	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑		↑			
Group	0			1			2		3			

b. The starting location of each subarray in the array

Values	98	32	200	75	1	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑		↑			
Group	0			1			2		3			

c. The value 75 is exchanged to the first position in group 1

Values	1	32	200	75	98	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑		↑			
Group	0			1			2		3			

d. The value 98 is exchanged to the second position in group 1

Values	1	32	25	75	98	88	125	137	200	170	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑		↑			
Group	0			1			2		3			

e. The array after all elements are moved to their groups

Values	1	25	32	75	88	98	125	137	165	170	188	200
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑		↑			
Group	0			1			2		3			

f. The final list

Figure 1. An example of using Groupsort

3.2 Flowchart for Groupsort

Based on the foregoing algorithm, an efficient implementation of Groupsort is presented in Figure 2, in which it is assumed that the values of the n keys are stored as positive integers in the array elements $X[1]$, ..., $X[n]$ and K groups are used.

3.3 Theoretical Running Time Analysis of Groupsort

From flowchart in Figure 2, we can analyze the average running time step by step.

In Step 1, we simply scan the input and find the minimum and maximum values. Therefore, the average running time of Step 1 is O(n).

In Step 2, the average running time is O(n) since it also requires scanning the input to find the size of each group and store it in array *Count*.

In Step 3, 4, and 5, the group number of each input element is computed and compared to the number of the group currently being processed. Every element not initially in the correct group is exchanged with the first not-in-place element in the destination group, and the corresponding entry of *Count* is updated. Combining the above, the average running time is O(n).

In Step 6, Quicksort is used for sorting each group. If the number of groups is selected as a fixed proportion of the list size, say $K = \rho n$, for some $\rho \in (0,1)$, the expected number of elements in each group is equal to $1/\rho$ (assuming that the input elements are uniformly distributed). Then, the average running time to sort each group is approximately $O((1/\rho)\log(1/\rho))$. Because there are $K = \rho n$ groups, the average total running time for Step 6 is O(n).

Therefore, the average total running time of Groupsort is O(n).

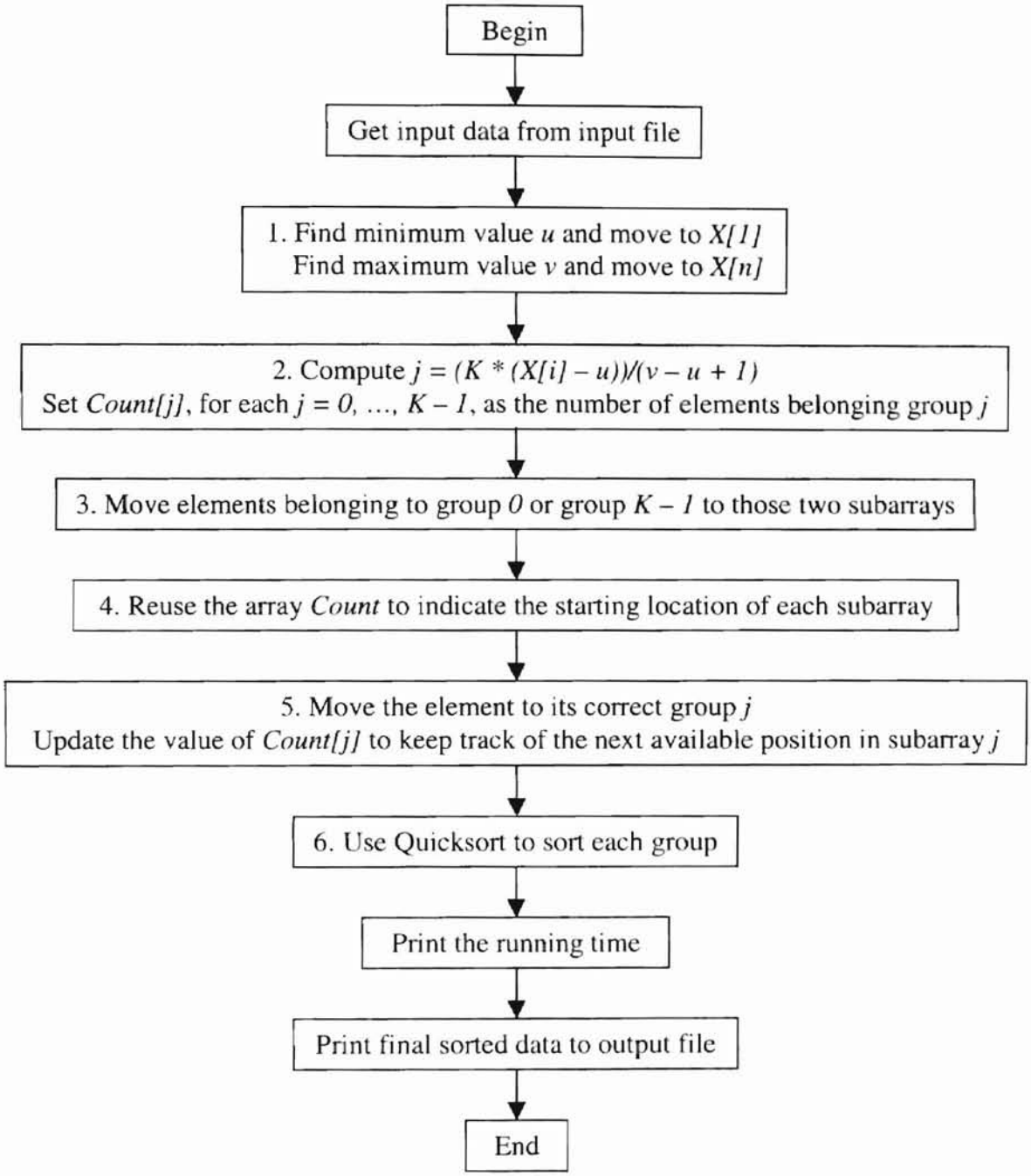


Figure 2. Flowchart for an efficient implementation of Groupsort

The worst case problem for Groupsort is that the sorted data are in a small range except one is far away. That means all data are in one group except one is in another group. Using Quicksort, the worst case time is $O(n^2)$.

3.4 The C Programming Code for Groupsort

The C programming code for Groupsort is attached in Appendix A. The C function qsort was used for the implementation of Quicksort. According to their documentation, this implementation uses the "median of three" rule for choosing the pivot element.

CHAPTER IV

FLASHSORT1

4.1 Sorting Algorithm

The following description of Flashsort1 algorithm is quoted from [21].

"The Flashsort1 algorithm consists of three logical blocks: classification, permutation, and straight insertion.

Classification determines the size of each class of elements. If the maximal element is *Amax* and the minimal element is *Amin*, we can compute:

$$K(A(i)) = 1 + INT((m-1)(A(i) - Amin) / (Amax - Amin))$$

The result will be a value from 1 to m, which is called the *class* of $A(i)$. After computing the actual number of elements in each class, we can predict where each class will appear in the final array. The vector L is used to track this information.

Permutation is used to move elements into the correct class. We simply compute each element's class index K and place it into the location indicated by $L(K)$. We then decrement $L(K)$. When the first class has been filled up, we need to begin the next permutation cycle. The process stops when we've moved every element.

Once the permutation step has moved everything into approximately the correct place, we have a partially sorted array to work with. Following Sedgewick [26], we use a straight insertion sort over the entire array."

Here is an example of using Flashsort1. Suppose there are 12 keys, stored in *A[1], ..., A[12]*, with maximal element *Amax* = 200 and minimal element *Amin* = 1, as shown in Figure 3a.

Suppose that the class number $K = 4$. Through passing the array in Figure 3a, we can compute the actual number of elements (3, 4, 4, and 1, respectively) falling in each class, as shown in Figure 3b, in which the pointer (arrow) $L[K]$ indicates the end position that will contain the elements in class K.

Next, we need move elements into the correct class. In this example, the first key, whose value is 75, belongs to class 2 and is therefore placed into the location indicated by $L[2] = 7$. We then decrement $L[2]$. And the evicted element 137 is held by a temporary variable *HOLD* as the next element that will be moved, as shown in Figure 3c. When the first class has been filled up, we need to determine the next permutation cycle. Through searching the array, we can find the element $A[j]$ that satisfies the condition $j<L(K(A(j)))$ as the next permutation cycle leader. In this example, the element is 98, as shown in Figure 3d. When the total number of moves is equal to the data number, in this example $MOVE = 12$, the permutation completes, as shown in Figure 3e.

Finally, straight insertion is used to sort each class to obtain the final list, as show in Figure 3f.

4.2 Flowchart for Flashsort1

Based on the foregoing algorithm, an implementation of Flashsort1 is presented in Figure 4, in which it is assumed that the values of the n keys are stored as positive integers in the array elements $A[1], \ldots, A[n]$ and we sort the array by use of index vector L of dimension M.

Values	75	32	200	98	1	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12

a. The original array to be sorted

Values	75	32	200	98	1	125	137	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
			↑				↑				↑	↑
Vector			*L[1]*				*L[2]*				*L[3]*	*L[4]*

b. The end location of each class in the array

Values		32	200	98	1	125	75	88	170	25	165	188
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
			↑			↑					↑	↑
Vector			*L[1]*			*L[2]*					*L[3]*	*L[4]*

c. The value 75 is moved to class 2 (*HOLD*=137, *MOVE*=1)

Values	32	1	25	98	125	88	75	170	188	165	137	200
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑			↑			↑				↑	
Vector	*L[1]*			*L[2]*			*L[3]*				*L[4]*	

d. The first class has been filled up (*HOLD*=98, *MOVE*=11)

Values	32	1	25	98	125	88	75	170	188	165	137	200
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑		↑				↑				↑	
Vector	*L[1]*		*L[2]*				*L[3]*				*L[4]*	

e. The array after all elements are moved to their classes (*MOVE*=12)

Values	1	25	32	75	88	98	125	137	165	170	188	200
Subscripts	1	2	3	4	5	6	7	8	9	10	11	12
	↑		↑				↑				↑	
Vector	*L[1]*		*L[2]*				*L[3]*				*L[4]*	

f. The final list

Figure 3. An example of using Flashsort1

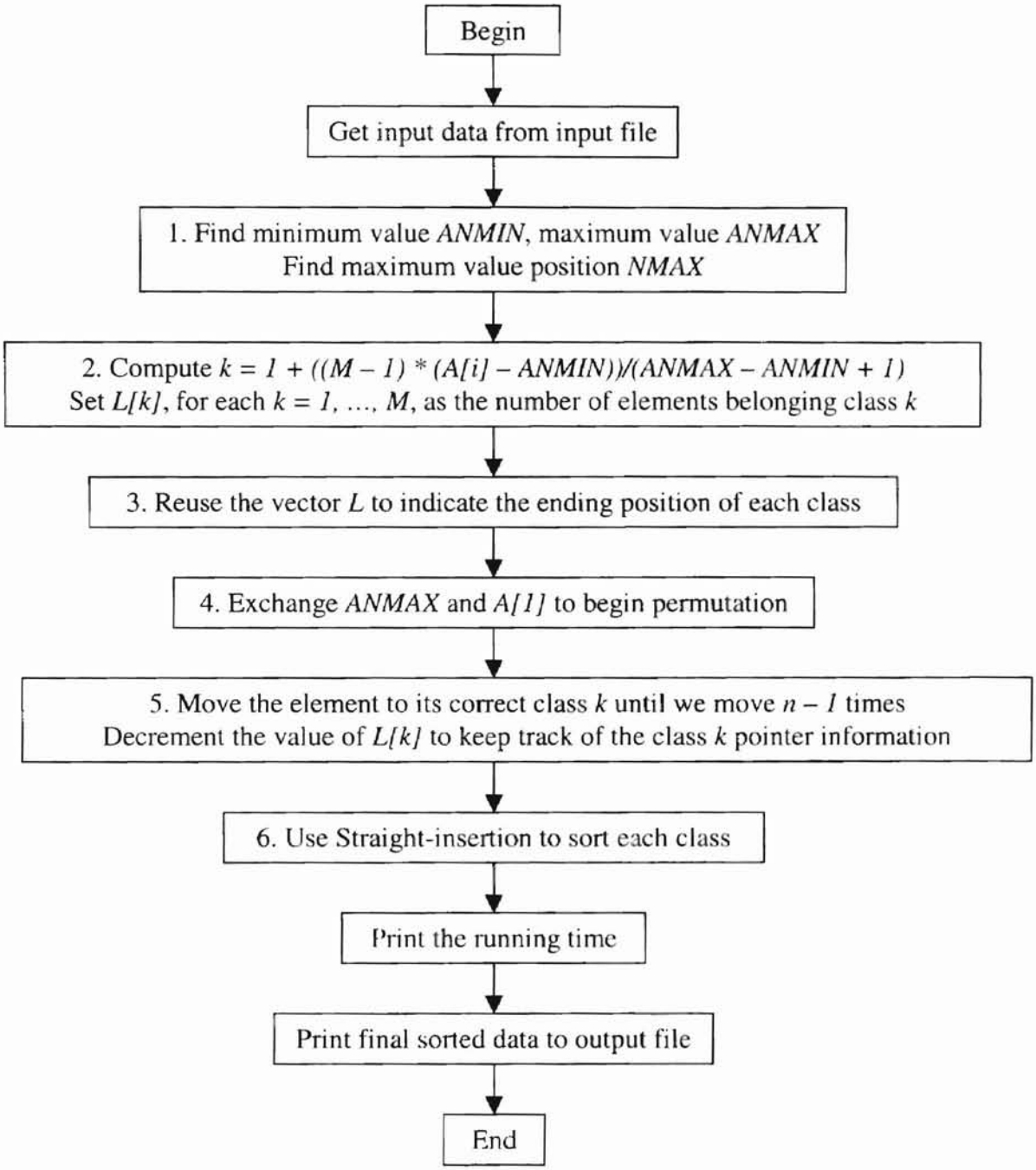


Figure 4. Flowchart for an implementation of Flashsort1

4.3 Theoretical Running Time Analysis of Flashsort1

From flowchart in Figure 4, we can analyze the average running time step by step.

In Step 1, we simply scan the input and find the minimum and maximum values. Therefore, the average running time of Step 1 is O(n).

In Step 2, the average running time is O(n) since it also requires scanning the input to find the size of each class and store it in index vector L.

In Step 3, 4, and 5, we move input elements into the correct class. Each item is moved exactly once. We can simply count the total number of moves and stop when we've moved every element. Therefore, the average running time is O(n).

In Step 6, a straight insertion sort is used for sorting each class. If the number of classes is selected as a fixed proportion of the list size, say $m = \rho n$, for some $\rho \in (0,1)$, the expected number of elements in each class is equal to $1/\rho$ (assuming that the input elements are uniformly distributed). Then, the average running time to sort each class is approximately $O((1/\rho)^2)$. Because there are $m = \rho n$ classes, the average total running time for Step 6 is O(n).

Therefore, the average total running time of Flashsort1 is O(n).

The worst case problem for Flashsort1 is that the sorted data are in a small range except one is far away. That means all data are in one class except one is in another class. Using straight insertion sort, the worst case time is $O(n^2)$.

4.4 The C Programming Code for Flashsort1

The C programming code for Flashsort1 is attached in Appendix B.

CHAPTER V

COMPUTATIONAL RESULTS

This chapter contains computational results for the relative running time of Groupsort, Flashsort1, bucket sort, and Quicksort. The C programming code for bucket sort is attached in Appendix C. The C programming code for Quicksort using qsort function is attached in Appendix D. Appendix E shows the C programming code that is used to generate uniformly distributed input integers. And Appendix F shows the C programming code that is used to generate truncated normal distributed input integers.

The tests are performed on a personal computer with an Intel 82443 LX/FX Pentium® II Processor. Two test series are performed.

5.1 Run Time Results for Different Additional Storage Space

In the first series, Groupsort and Flashsort1 are tested against bucket sort using uniformly distributed integer values in the range 1 to 32767, for list sizes n varying from 2000 to 25000. A number of different additional storage spaces are used with values of $0.05n$, $0.1n$, $0.2n$, and $0.3n$. For each list size n and each kind of additional storage space used, 15 arrays are generated and sorted. Table 2 shows the average relative running times.

5.2 Run Time Results for Different Data Distribution

The second series of tests is designed to determine how the efficiency of Groupsort and Flashsort1 is affected by input data distribution, in comparison to

Quicksort. To this end, the input array elements are generated according to a normal distribution truncated in the range 1 to 32767, with mean $\mu = 16384$ and standard deviation σ varying from 2 to 10000. Smaller values of σ imply a higher degree of nonuniformity [20]. A number of different standard deviations are used with values of 2, 10, 1000, and 10000. For each list size n and standard deviation σ, 15 arrays are generated and sorted. Table 3 shows the average relative running times.

Table 2. Relative running times under uniformly distributed integer array

List size n	Groupsort	Flashsort1	Bucket sort
2000	67	40	87
5000	180	133	231
10000	367	260	560
15000	553	420	833
20000	700	573	1060
25000	900	673	1367

a. additional storage space $0.05n$

List size n	Groupsort	Flashsort1	Bucket sort
2000	80	47	87
5000	187	133	200
10000	333	247	427
15000	513	360	647
20000	673	473	840
25000	833	600	1060

b. additional storage space $0.1n$

List size n	Groupsort	Flashsort1	Bucket sort
2000	73	33	90
5000	167	107	200
10000	320	207	380
15000	473	347	527
20000	613	440	727
25000	813	567	940

c. additional storage space $0.2n$

List size n	Groupsort	Flashsort1	Bucket sort
2000	53	33	80
5000	167	100	200
10000	307	207	350
15000	453	320	553
20000	640	453	700
25000	780	547	907

d. additional storage space $0.3n$

Table 3. Relative running times under truncated normal distributed integer array

List size n	Groupsort	Flashsort1	Quicksort
2000	397	1227	393
5000	2627	7267	2620
10000	10547	29547	10513
15000	24627	67047	23980
20000	43700	120993	43220
25000	65700	195960	64800

a. standard deviation $\sigma = 2$

List size n	Groupsort	Flashsort1	Quicksort
2000	80	727	73
5000	433	6247	440
10000	1580	23280	1847
15000	2533	35733	4173
20000	3853	52360	7600
25000	5167	65267	11840

b. standard deviation $\sigma = 10$

List size n	Groupsort	Flashsort1	Quicksort
2000	73	33	53
5000	173	107	186
10000	360	233	413
15000	527	373	600
20000	733	480	833
25000	873	613	1067

c. standard deviation $\sigma = 1000$

List size n	Groupsort	Flashsort1	Quicksort
2000	73	40	60
5000	187	120	187
10000	360	227	407
15000	513	353	587
20000	673	500	853
25000	867	613	1080

d. standard deviation $\sigma = 10000$

CHAPTER VI

COMPARISON OF GROUPSORT AND FLASHSORT1

As mentioned in chapter 1, we consider three aspects that affect the performance of an algorithm: running time, additional storage space, and data distribution. This chapter compares Groupsort and Flashsort1 with bucket sort and Quicksort on all these aspects.

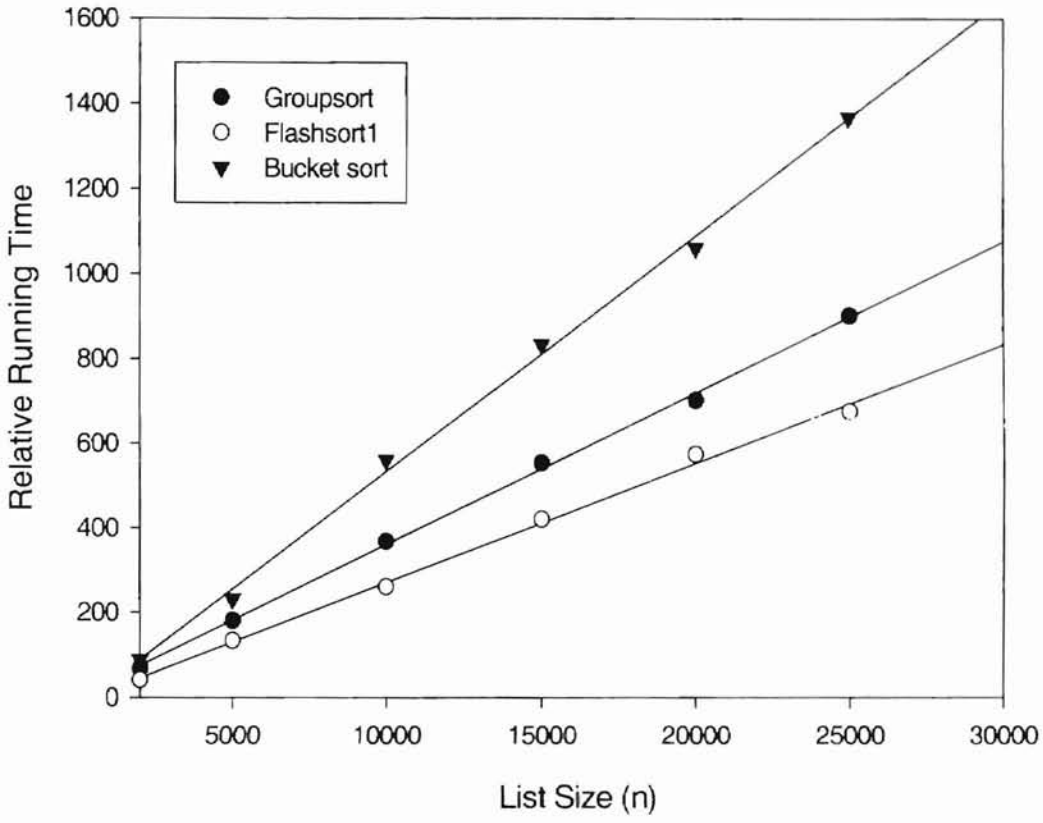
6.1 Running Time and Their Impact

The empirical results on the relative running time with different list sizes are presented in Figure 5, 6, 7, and 8. Four cases are examined with different additional storage spaces: $0.05n$, $0.1n$, $0.2n$, and $0.3n$.

For smaller list sizes, all methods require negligible times. The results indicate that for all values $n >= 2000$, Flashsort1 is faster than Groupsort, and Groupsort is faster than bucket sort. For example, $n = 20000$, additional storage space $0.1n$, Flashsort1 is 44% faster than bucket sort. Groupsort is 20% faster than bucket sort.

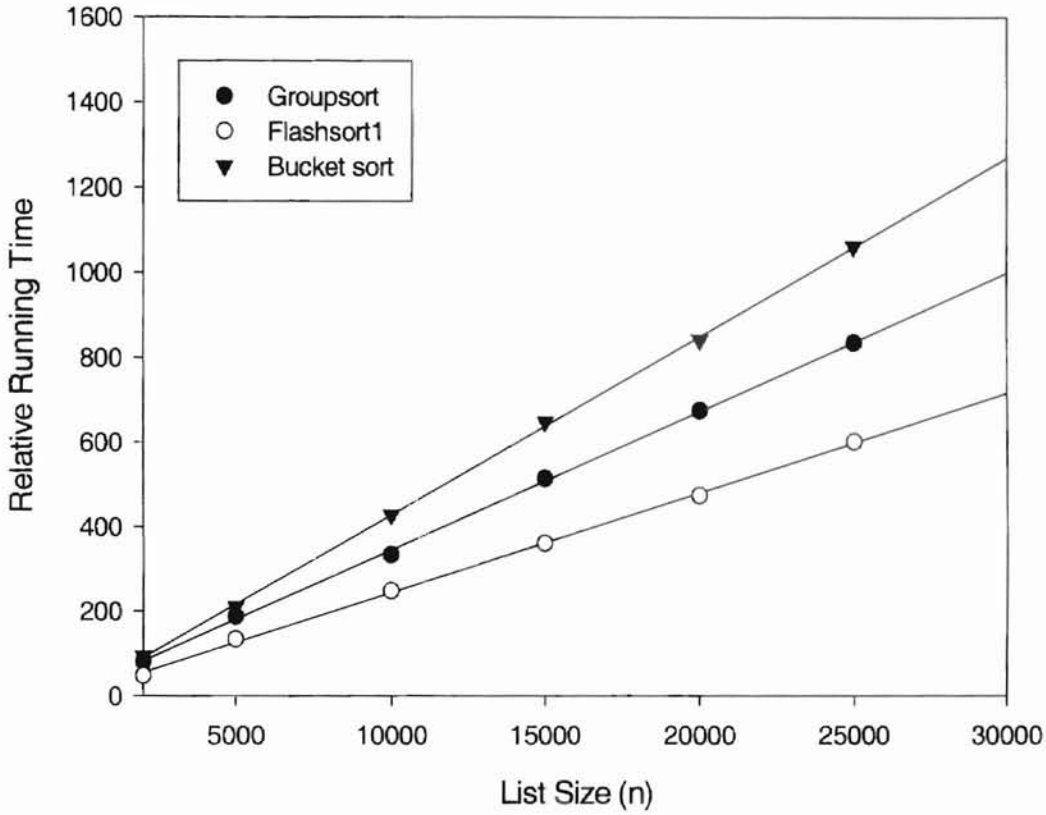
From the fit curve function y = ax + b, we can see, for Flashsort1, a is in the range 0.024 to 0.028; for Groupsort, a is in the range 0.031 to 0.036; for bucket sort, a is in the range 0.035 to 0.056. These results indicate that the running time increases with list size as the following sequence: Flashsort1 < Groupsort < bucket sort. The bigger the size of sorted list, the more efficient is Flashsort1 relative to Groupsort.

From the running time view, Flashsort1 has better performance than Groupsort and Groupsort has better performance than bucket sort.



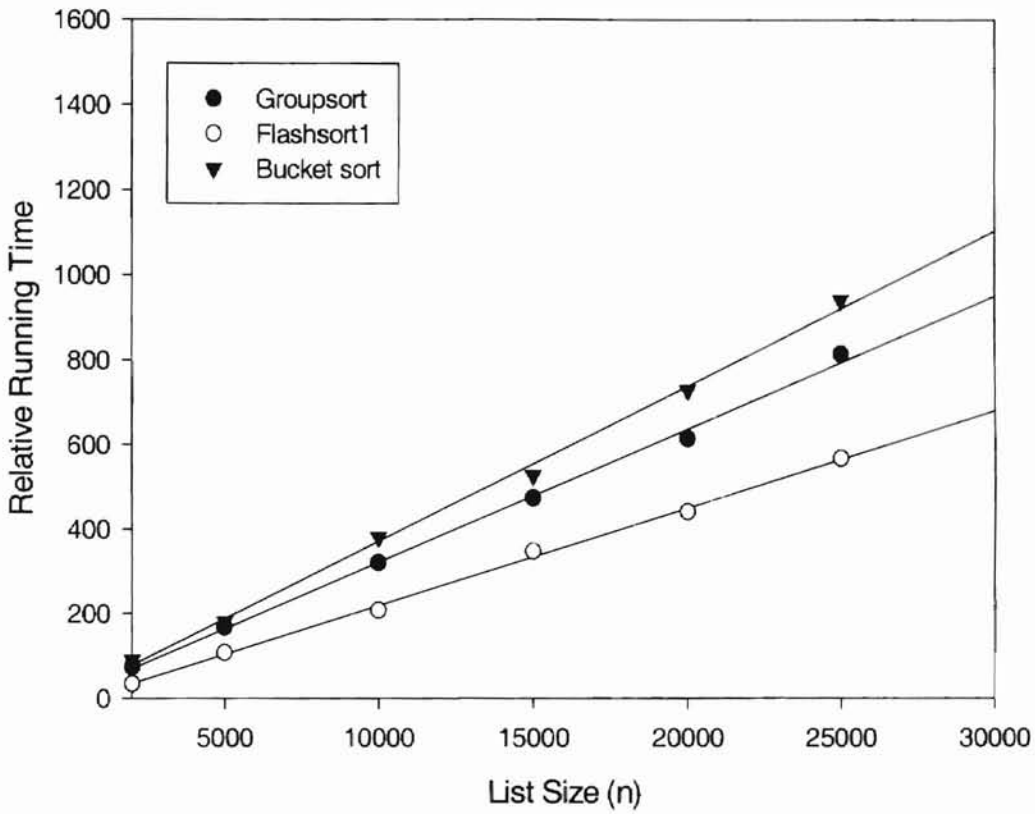
Fitted function: Groupsort: y = 0.036x + 1.821
Flashsort1: y = 0.028x – 11.125
Bucket sort: y = 0.056x – 22.696

Figure 5. Relative running times with list size for additional storage space $0.05n$



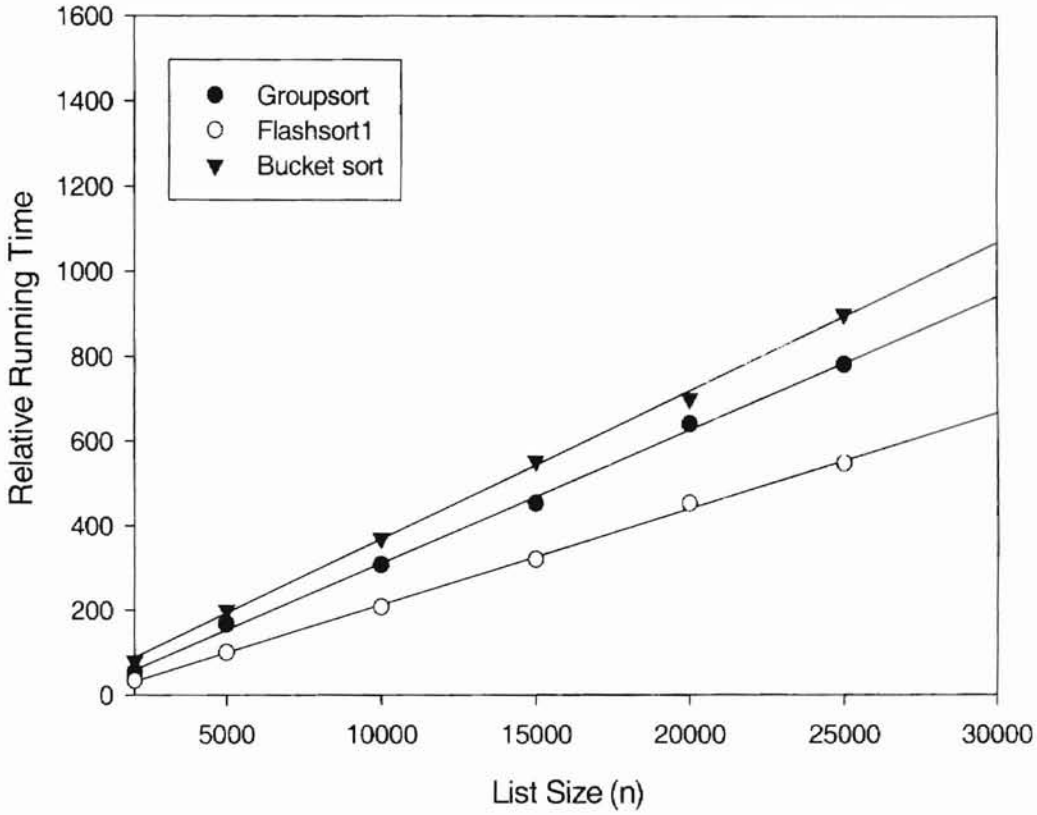
Fitted function: Groupsort: y = 0.033x + 16.382
Flashsort1: y = 0.024x + 7.287
Bucket sort: y = 0.042x + 5.815

Figure 6. Relative running times with list size for additional storage space $0.1n$



Fitted function: Groupsort: y = 0.031x + 5.827
Flashsort1: y = 0.023x – 12.728
Bucket sort: y = 0.036x + 3.561

Figure 7. Relative running times with list size for additional storage space $0.2n$



Fitted function: Groupsort: y = 0.031x – 4.078
Flashsort1: y = 0.023x – 14.510
Bucket sort: y = 0.035x + 17.869

Figure 8. Relative running times with list size for additional storage space $0.3n$

6.2 Additional Storage Space and Their Impact

The empirical results on the relative running time with different additional storage space are presented in Figure 9, 10, 11, 12, 13, and 14. Six cases are examined with different list size: 2000, 5000, 10000, 15000, 20000, and 25000.

The running time depends on two sections: Step 1 to 5 in Figure 2 for Groupsort or in Figure 4 for Flashsort1, and Step 6 in Figure 2 for Groupsort or in Figure 4 for Flashsort1 in which a conventional comparison sorting algorithm (Quicksort for Groupsort, straight insertion for Flashsort1) is used to sort each group or class. From the theoretical view, increasing the additional storage space will benefit Groupsort and Flashsort1 by sorting smaller groups or classes (under uniformly distributed assumption) in Step 6.

But in the actual tests, this is not always the case. As shown in Figure 9 and Figure 10, we can see that for relative small list size, say $n = 2000$ and $n = 5000$, when the additional storage space is $0.05n$, Groupsort and Flashsort1 both have better performance than when the additional storage space is $0.1n$. The reason is when the sorted list size is relative small and the additional storage space is relative small, increasing a little bit additional storage space may not always benefit the sorting.

For large list size, say $n = 10000$, $n = 15000$, $n = 20000$, and $n = 25000$, as shown in Figure 11 to Figure 14, running times are decreasing with increasing additional storage space. Table 4 shows the running speed comparison with additional storage space $0.1n$. When we increase the additional storage space from $0.1n$ to $0.2n$ and $0.3n$ which means we double the additional storage space, the best improvement is only 12% for Groupsort and 16% for Flashsort1. When we increase the additional storage space from $0.05n$ to

$0.1n$ which means we only use $0.05n$ more additional storage space, the best improvement can be 10% for Groupsort and 17% for Flashsort1. So, additional storage space $0.1n$ is the best choice for both Groupsort and Flashsort1 to achieve the best space/time trade-off.

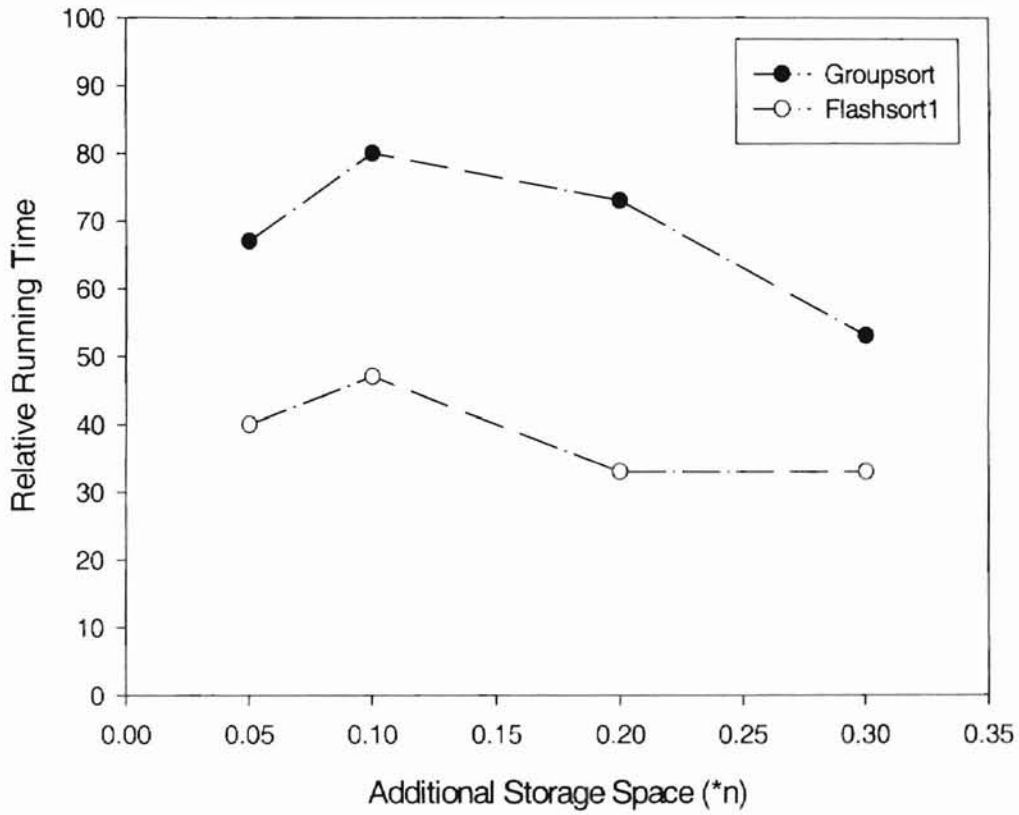


Figure 9. Relative running times with additional storage space for $n = 2000$

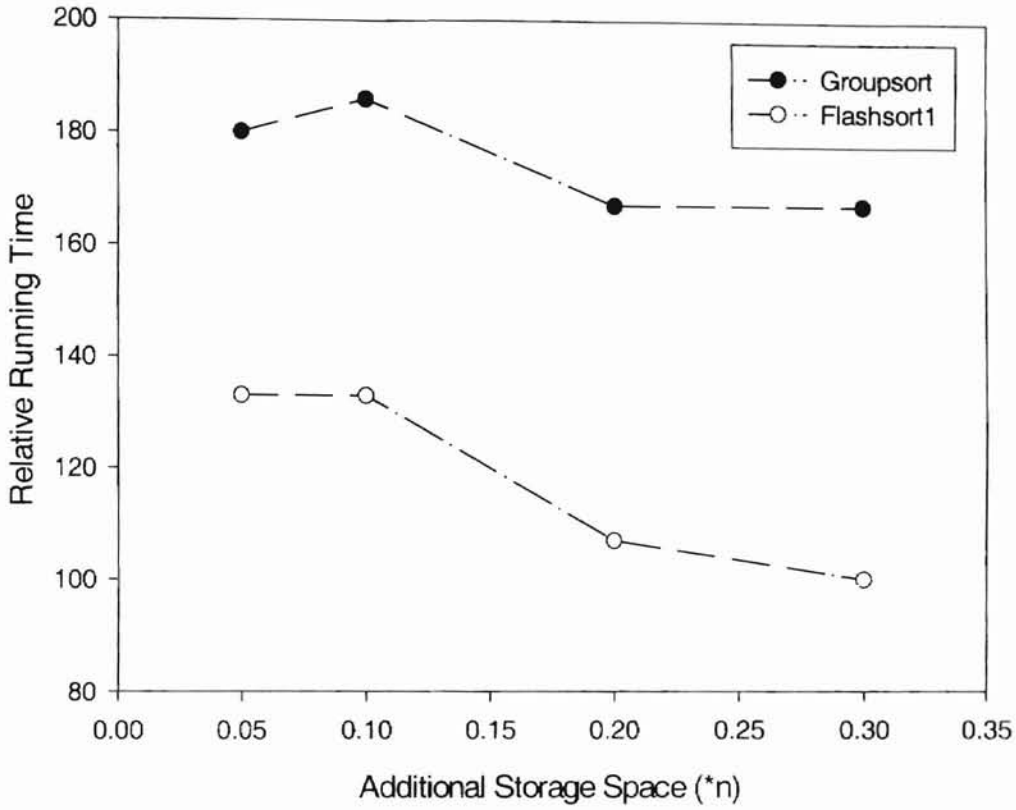


Figure 10. Relative running times with additional storage space for $n = 5000$

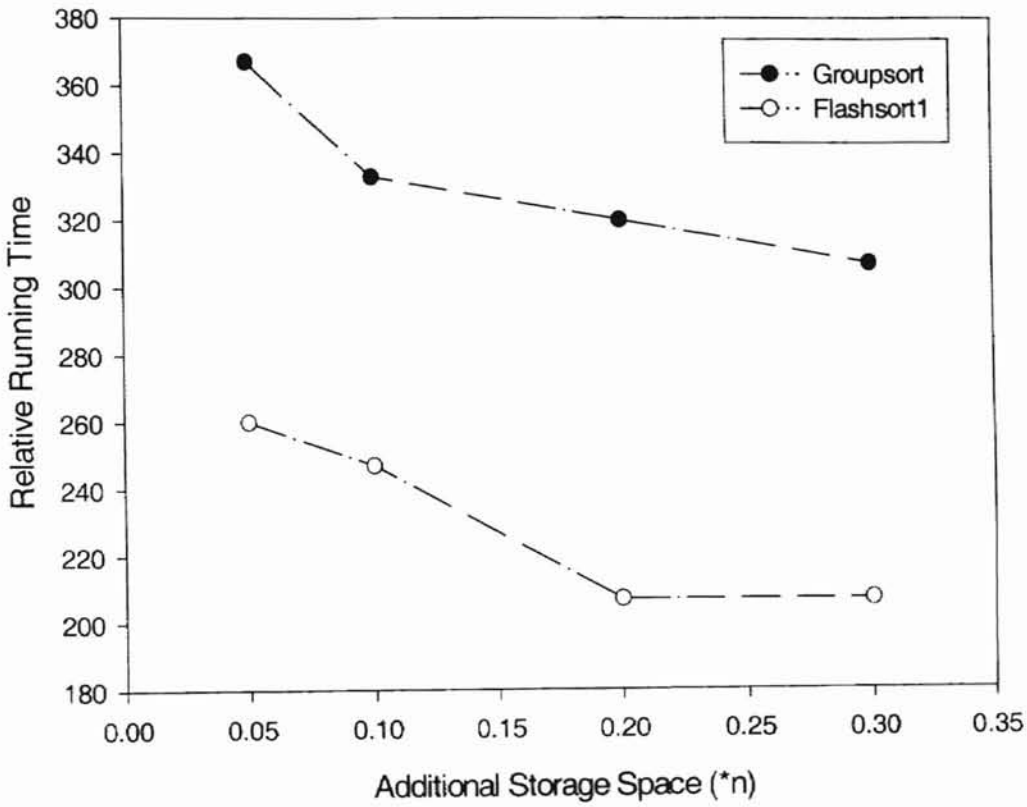


Figure 11. Relative running times with additional storage space for $n = 10000$

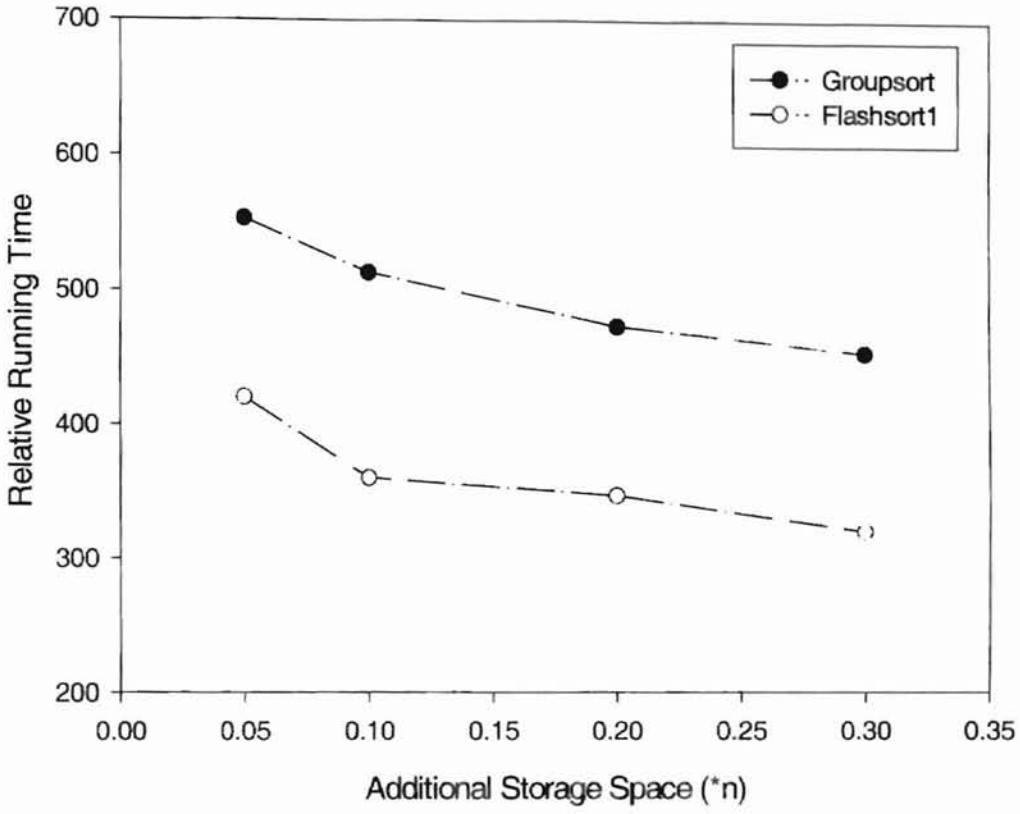


Figure 12. Relative running times with additional storage space for $n = 15000$

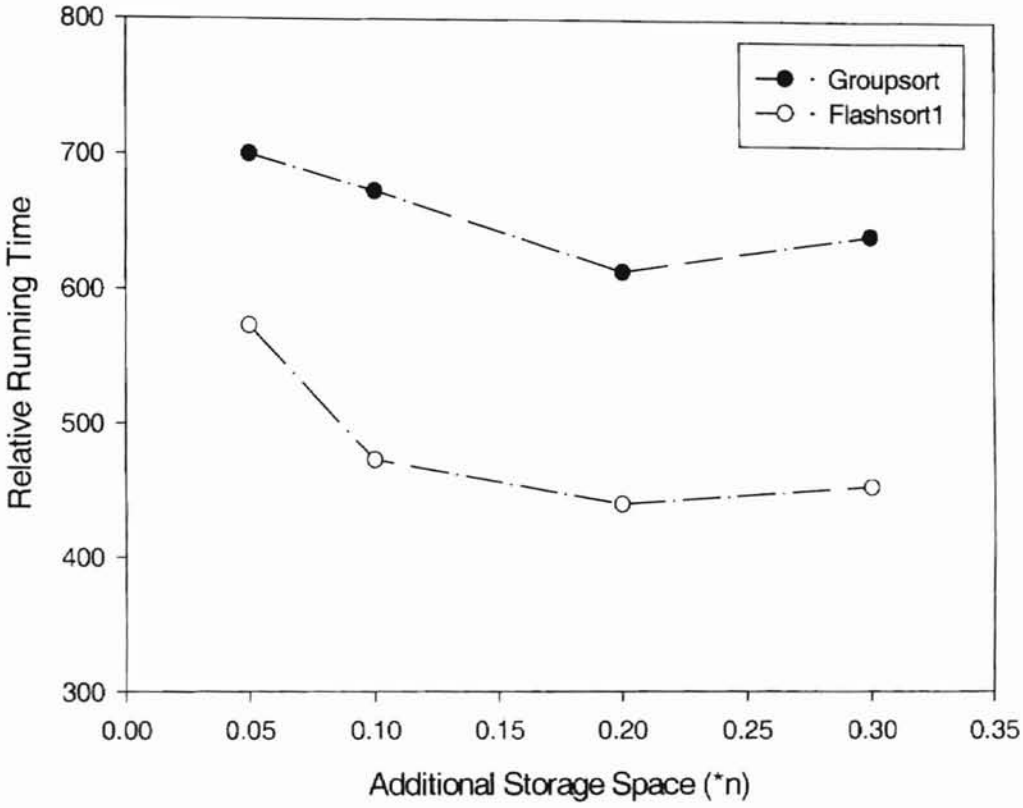


Figure 13. Relative running times with additional storage space for $n = 20000$

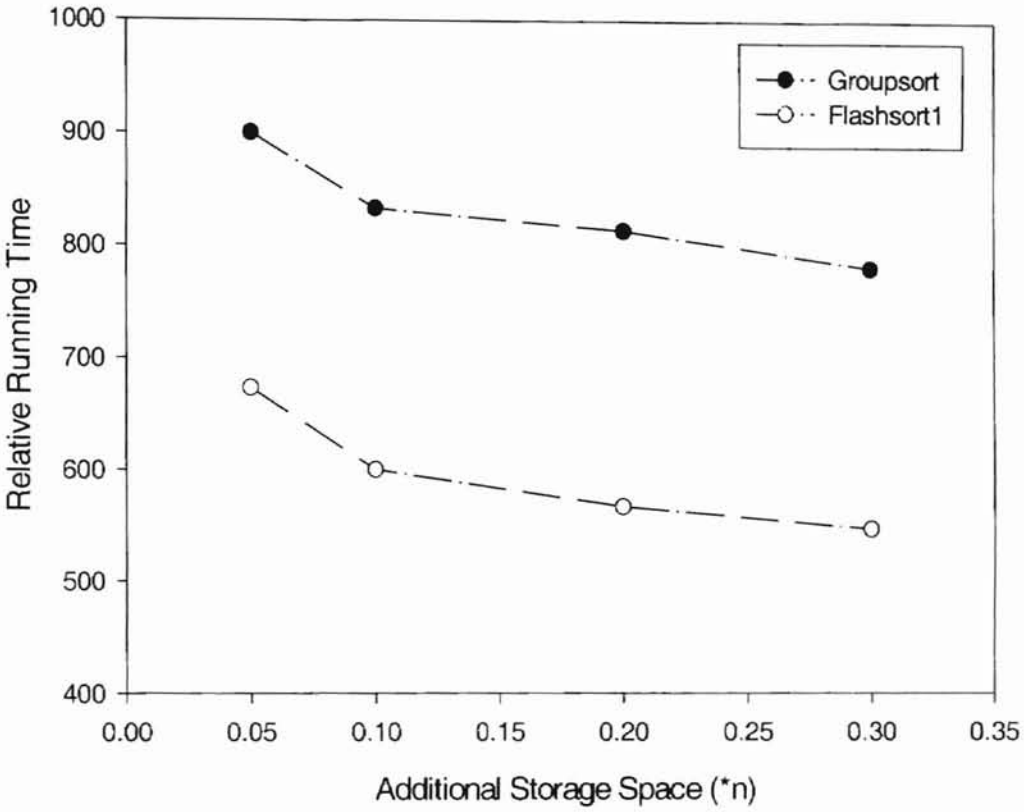


Figure 14. Relative running times with additional storage space for $n = 25000$

Table 4. Running speed comparison with additional storage space $0.1n$
(+: run faster, -: run slower)

List size n	$0.05n$	$0.2n$	$0.3n$
10000	- 10%	+ 4%	+ 8%
15000	- 8%	+ 8%	+ 12%
20000	- 4%	+ 9%	+ 5%
25000	- 8%	+ 2%	+ 6%

a. Groupsort

List size n	$0.05n$	$0.2n$	$0.3n$
10000	- 5%	+ 16%	+ 16%
15000	- 17%	+ 4%	+ 11%
20000	- 17%	+ 7%	+ 4%
25000	- 12%	+ 6%	+ 9%

b. Flashsort1

6.3 Data Distribution and Their Impact

The empirical results on the relative running time with different standard deviation are presented in Figure 15, 16, 17, 18, 19, and 20. Six cases are examined with different list size: 2000, 5000, 10000, 15000, 20000, and 25000 for additional storage space $0.1n$.

For standard deviation $\sigma = 1000$ and $\sigma = 10000$, Flashsort1 is faster than Groupsort and Quicksort, and Groupsort is faster than Quicksort for $n > 2000$. For standard deviation $\sigma = 2$ and $\sigma = 10$, the relative running times of Flashsort1 increase significantly. Flashsort1 runs more than 100 times slower than standard deviation $\sigma = 1000$ for $\sigma = 10$ and more than 300 times slower than standard deviation $\sigma = 1000$ for $\sigma = 2$. For standard deviation $\sigma = 2$ and $\sigma = 10$, the relative running times of Groupsort and Quicksort increase too. For better performance, we can use more efficient quicksort

algorithm instead of qsort function. For $\sigma = 10$, Groupsort is faster than Quicksort for $n > 2000$. And Groupsort is more efficient than Quicksort when the list size increases, as shown in Figure 16 to Figure 20. For $\sigma = 2$, Groupsort is slower than Quicksort.

From the above, we can see that Flashsort1 is much slower than Groupsort and Quicksort for standard deviation $\sigma = 2$ and $\sigma = 10$. And Groupsort is faster than Quicksort for $\sigma >= 10$ when $n > 2000$. When $\sigma = 2$, Groupsort is slower than Quicksort.

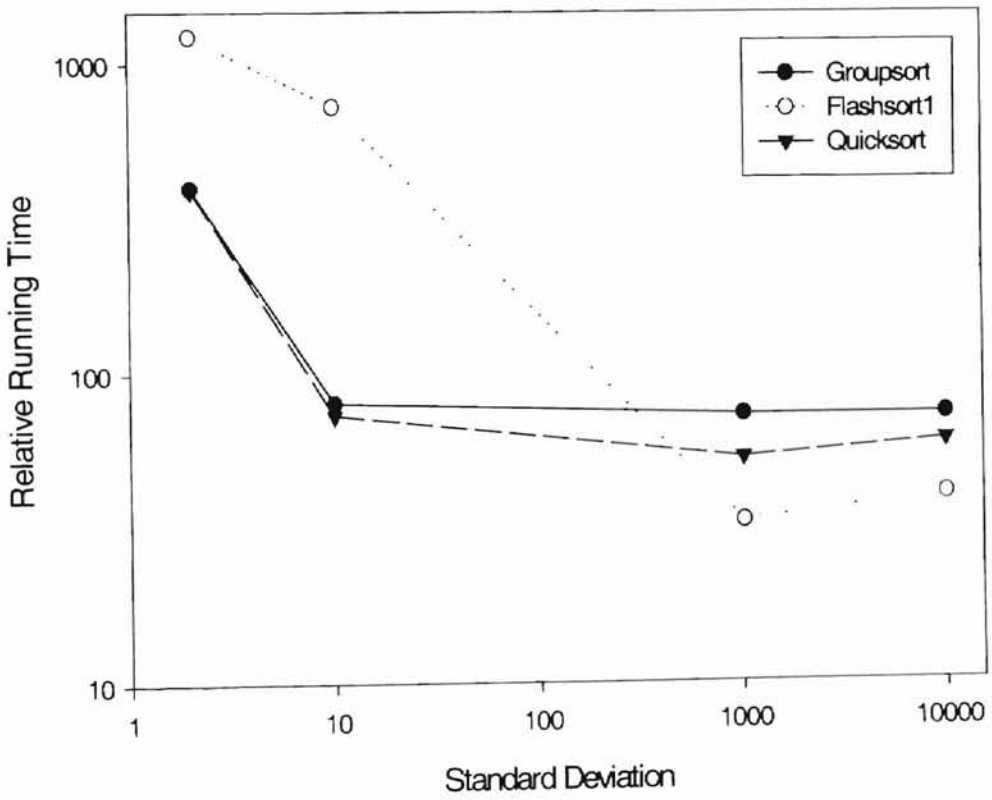


Figure 15. Relative running times with different standard deviations for $n = 2000$

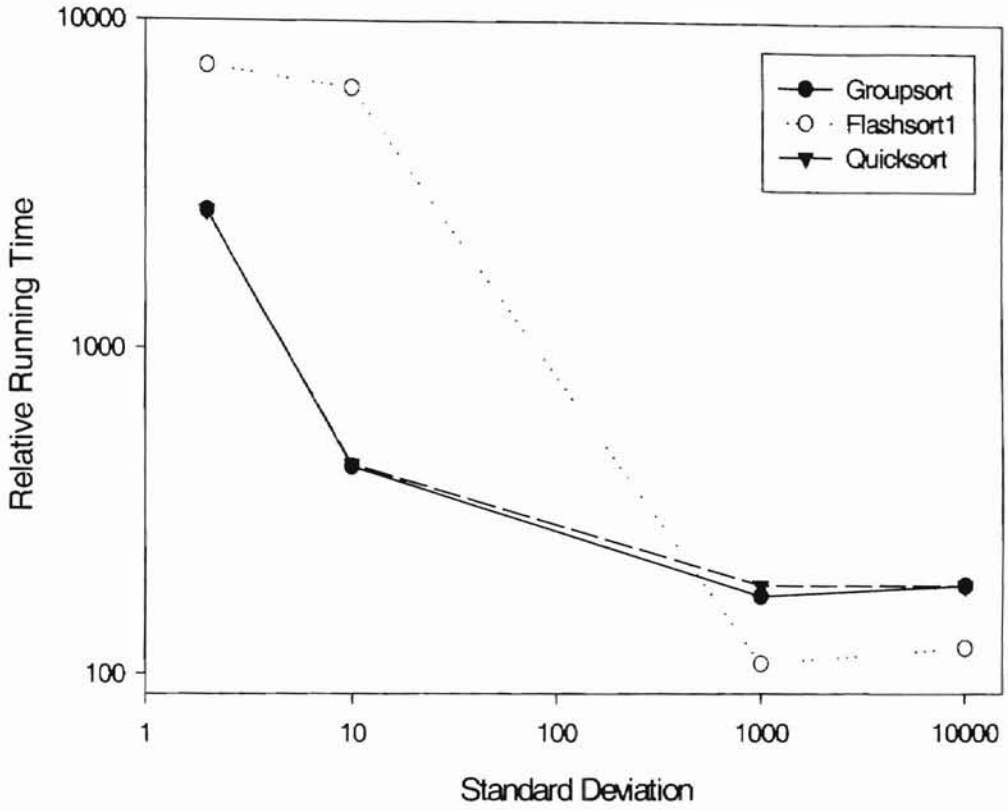


Figure 16. Relative running times with different standard deviations for $n = 5000$

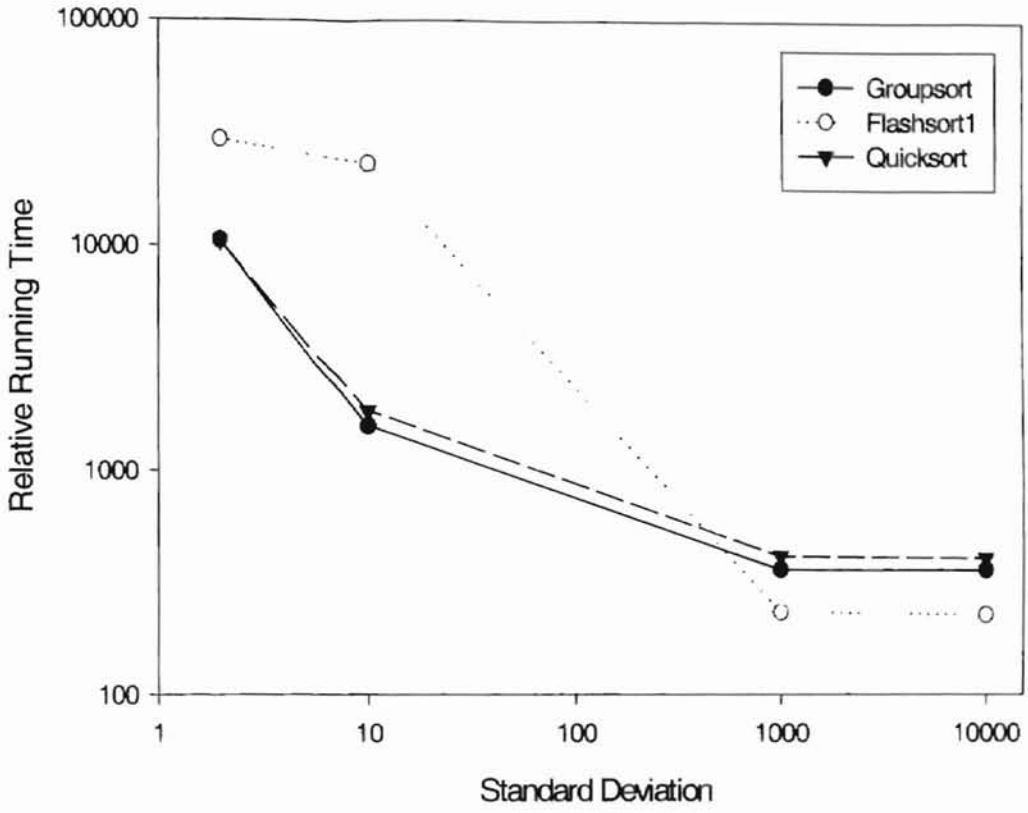


Figure 17. Relative running times with different standard deviations for $n = 10000$

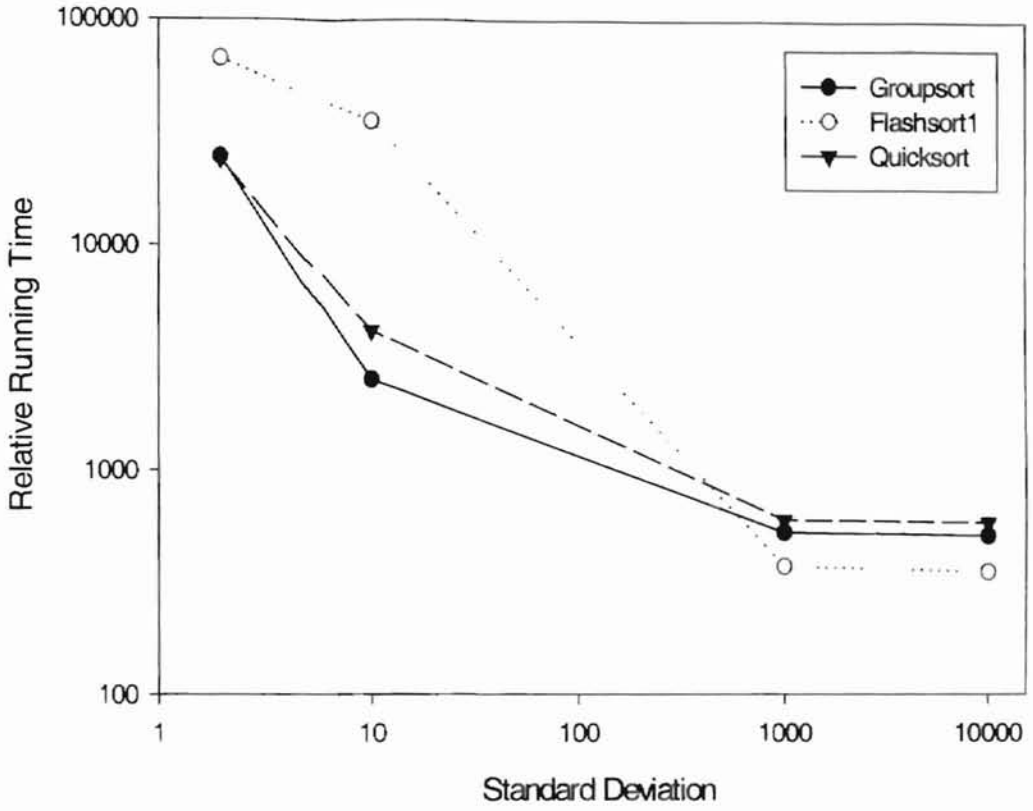


Figure 18. Relative running times with different standard deviations for $n = 15000$

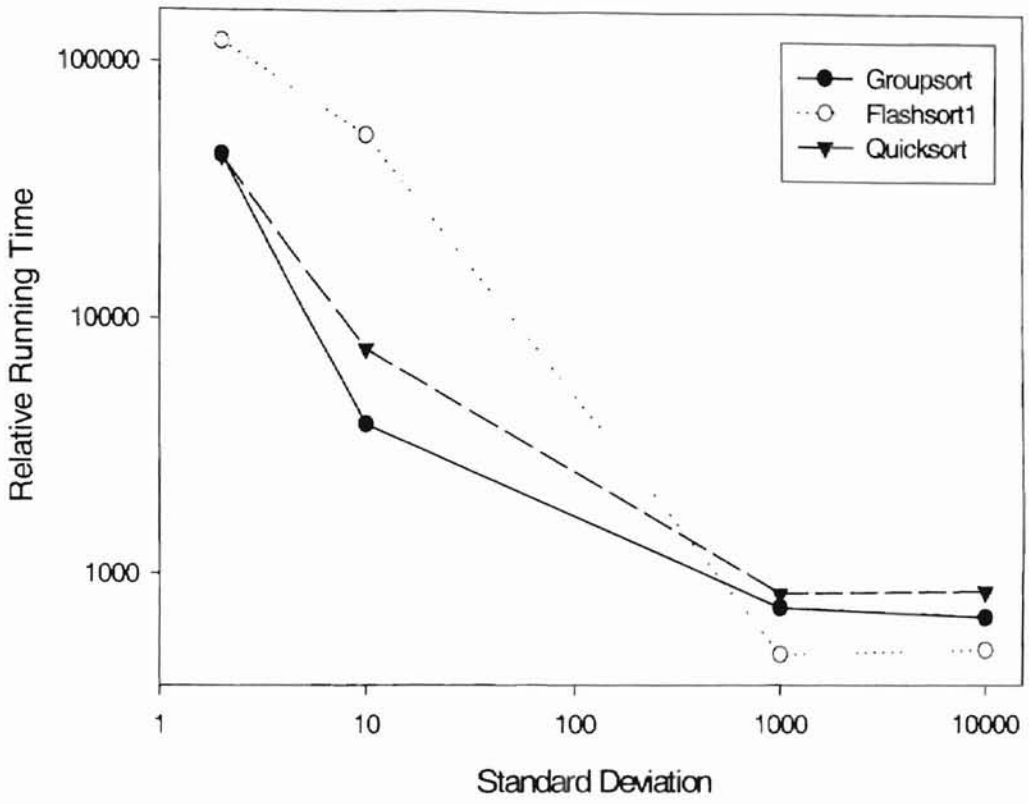


Figure 19. Relative running times with different standard deviations for $n = 20000$

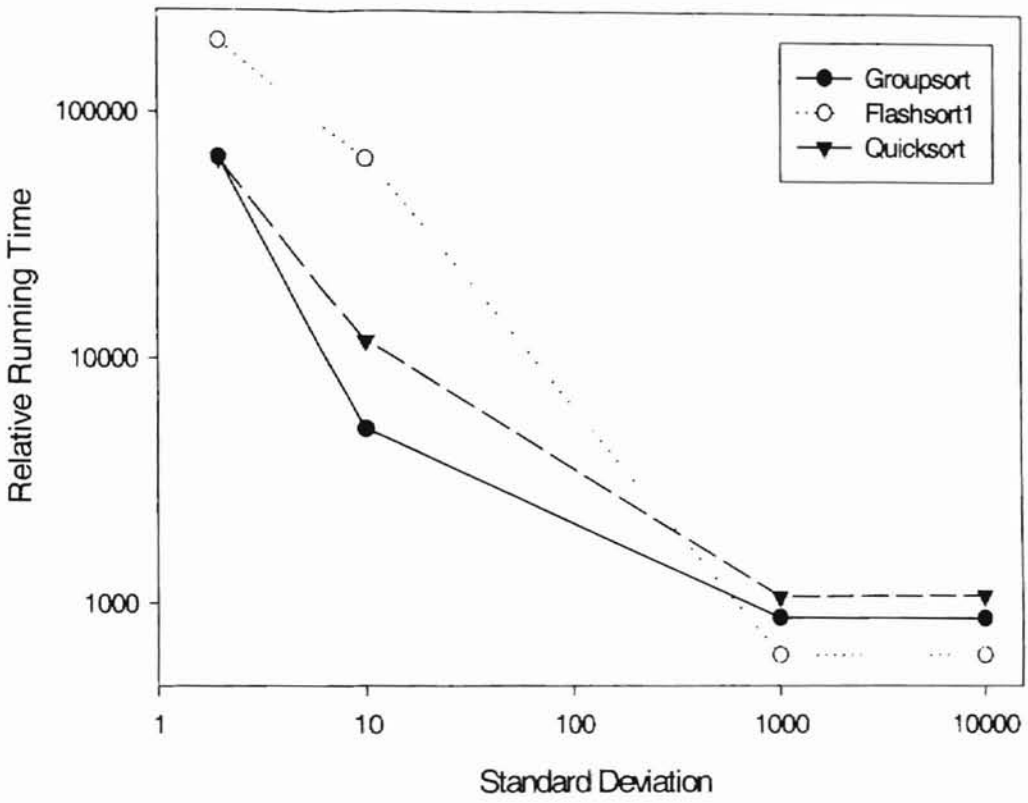


Figure 20. Relative running times with different standard deviations for $n = 25000$

CHAPTER VII

CONCLUSIONS AND IMPROVEMENTS

7.1 Conclusions

As discussed in last chapter, we can see that Groupsort and Flashsort1 both have linear average running time and run faster than bucket sort under the assumption of a uniformly distributed list. The optimal additional storage space for these more efficient sorting algorithms is $0.1n$ for a large list ($n \geq 10000$), or $0.05n$ for a relatively small list ($n = 2000$ and $n = 5000$).

Under the assumption of a uniformly distributed list, Flashsort1 is always faster than Groupsort with the same additional storage space. So Flashsort1 is a better choice for sorting uniformly distributed data.

Flashsort1 is highly affected by the data distribution. Flashsort1 should be chosen only when $\sigma >= 1000$. If the degree of nonuniformity is very high (say $\sigma <= 10$), Groupsort is much better than Flashsort1. And for $\sigma >= 10$, Groupsort is more efficient than Quicksort.

7.2 Improvements

Flashsort1 is a better choice for sorting uniformly distributed data. But it is very bad for sorting data that are very nonuniformly distributed. To make Flashsort1 perform better under a truncated normal distributed list, we can change Step 6 in the Flashsort1 flowchart in Figure 4 from Straight-insertion to Quicksort, even to Heapsort, to sort each class.

BIBLIOGRAPHY

1. S. Baase, *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, MA, 58-132 (1978).
2. K. E. Batcher, Bitonic Sorting. *Goodyear Aerospace Rep. GER-11869* (1964).
3. M. A. Batty, Certification of Algorithm 201: Shellsort. *Comm. ACM*, 7(6), 349 (1964).
4. B. L. Bauer, *An Empirical Study of Shellsort, Oklahoma State University M.S. Thesis.* (1980)
5. C. F. Bowman, *Algorithms and Data Structures: An Approach in C*. Harcourt Brace College Publishers, Fort Worth, (1994).
6. A. Burnetas, D. Solow and R. Agarwal, An analysis and implementation of an efficient in-place bucket sort. *Acta Informatica*, 34, 687-700 (1997).
7. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*. MIT Press, (1990).
8. F. Ducoin, Tri par addressage direct. *R.A.I.R.O. Informatique/Computer Science*, 13(3), 225-237 (1979).
9. R. W. Floyd, Algorithm 245: TREESORT3. *Comm. ACM*, 7, 701 (1964).
10. E. Gamson and C. Picard, Algorithme de tri par addressage direct. *C. R. Acad. Sc. Paris 269, Serie A*, 38-41 (1969).
11. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures In Pascal and C, Second Edition*. Addison-Wesley, Reading, MA, 154-155 (1991).
12. B. C. Huang and D. E. Knuth, A one-way, stackless quicksort algorithm. *BIT*, 26, 127-130 (1986).

13. C. A. R. Hoare, Algorithm 64: Quicksort. *Comm. ACM*, 4(7), 321 (1961).

14. C. A. R. Hoare, Quicksort. *Computer J.*, 5(1), 10-15 (1962).

15. C. A. R. Hoare, *Essays in Computer Science*. Prentice-Hall, (1992).

16. C. A. R. Hoare, Algorithm 63: Partition and 64: Quicksort. *Comm. ACM*, 4(7), 321 (1961).

17. O. C. Juelich, Remark on algorithm 175: Shuttlesort. *Comm. ACM*, 6(12), 739 (1963).

18. O. C. Juelich, Remark on algorithm 175: Shuttlesort. *Comm. ACM*, 7(5), 296 (1964).

19. D. E. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching, Second Edition.* Addison-Wesley, Reading, MA, (1997).

20. J. P. Linderman, Theory and practice in the construction of a working sort routine. *Bell System Technical Journal*, 63, 1827-1843 (1984).

21. K. D. Neubert, The Flashsort1 algorithm. *Dr. Dobb's Journal*, February, 123-125 (1998).

22. V. R. Pratt, *Shellsort and Sorting Networks.* Garland Publishing, New York, (1979).

23. S. M. Ross, *A First Course in Probability, Fourth Edition.* Macmillan College Publishing, New York, 210-222 (1994).

24. R. Sedgewick, *Algorithms*. Addision-Wesley, Reading, MA, (1988).

25. R. Sedgewick, *Quicksort, Stanford University Ph.D. Dissertation*. (1975).

26. R. Sedgewick, Implementing quicksort programs. *Comm. ACM*, 21(10), 847-856 (1978).

27. R. Sedgewick, The analysis of quicksort programs. *Acta Informatica*, 7, 327-355 (1977).

28. G. R. Schubert, Certification of algorithm 175: Shuttlesort. *Comm. ACM*, 6(10), 619 (1963).

29. C. J. Shaw and T. N. Trimble, Algorithm 175: Shuttlesort. *Comm. ACM*, 6(6), 312-313 (1963).

30. D. L. Shell, A high speed sorting procedure. *Comm. ACM*, 2(7), 30-32 (1959).

31. D. F. Stubbs and N. W. Webre, *Data Structures with Abstract Data Type and Ada*. Prindle, Weber, Schmidt-Kent, 301-341 (1993).

32. J. W. J. Williams, Algorithm 232: Heapsort. *Comm. ACM*, 7(6), 347-348 (1964).

APPENDIXES

APPENDIX A

C PROGRAMMING CODE FOR GROUPSORT

```
//Groupsort:
//Sorted date type is integer with range 1 to 32767
//Read sorted data from input file and put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50000  //Maximum input array size

//Global variables
int X[MAX+1];  //Sorted data array
int Count[MAX];  //Group array
int N; //Number of sorted data
int K; //Additional storage space size
int C;  //constant factor
int KK;  //Group number

//Input and output file pointer
FILE *ip;
FILE *op;

//Function declaration
void get_data(FILE *ip);
int find_u();
int find_v();
void set_Count(int u, int v);
void move_group(int u, int v);
void group(int u, int v);
static int test(int *x, int *y);
void printout();


//main function
main(int argc, char *argv[])
{
   int u, v, i;
   clock_t start_time, end_time;

   if(argc != 3)
   {
      printf("Use 'a.out inputfile outputfile' to run\n");
      exit(0);
   }

   if((ip=fopen(argv[1], "r"))==NULL)  //Open input file
   {
      printf("Cannot open the input file! Try again!\n");
      exit(0);
   }
```

```
   if((op=fopen(argv[2], "w"))==NULL)  //Open output file
   {
      printf("Cannot open the output file! Try again!\n");
      exit(0);
   }

   get_data(ip);  //Read data from input file
   start_time=clock();  //Get starting time
   u=find_u();  //Find minimum value
   v=find_v();  //Find maximum value
   C=(int)((v-u+1)/K)+1;
   set_Count(u, v);  //Find the number of elements in each group
   move_group(u, v);  //Move elements to the first and the last group
   group(u, v);  //Move elements to their correct group

   qsort(&X[0],Count[0]+1,sizeof(X[1]),test);  //Sort the first group

   for(i=1; i<=KK; i++)  //Sort other groups
   {
      qsort(&X[Count[i-1]+1],Count[i]-Count[i-1],sizeof(X[1]),test);
   }

   end_time=clock();  //Get ending time
   printf("The run time is %d\n", end_time-start_time);
   fprintf(op, "The run time is: %d\n", end_time-start_time);
   printout();  //Print the result to output file

   fclose(ip);
   fclose(op);
}

//Read data from input file
//Input file contains sorted data separated by blank space
void get_data(FILE *ip)
{
   int input;
   int count;
   int i;
   char temp[10];

   while(!feof(ip))
   {
      X[0]=0;

      for(count=1; fscanf(ip, "%d", &input)==1; count++)
      {
         if(count>MAX)
         {
            printf("Array is not enough to hold the elements to be
sorted!\n");
            exit(0);
         }

         X[count]=input;  //Put data into sorted data array
      }
   }
```

```
    N=count-1;  //Number of sorted data
    printf("The sorting array size is: %d\n", N);
    printf("Please enter the additional storage space size: ");
    K=atoi(gets(temp));  //Size of additional storage space

    fprintf(op, "The sorting array size is: %d\n", N);
    fprintf(op, "The additional storage space is: %d\n", K);
}

//Find the minimum value
int find_u()
{
    int u=X[1];
    int uposition=1;
    int temp,i;

    for(i=2; i<=N; i++)
    {
        if(X[i]<u)
        {
            u=X[i];
            uposition=i;
        }
    }

    temp=X[1];
    X[1]=u;
    X[uposition]=temp;

    return u;
}

//Find the maximum value
int find_v()
{
    int v=X[N];
    int vposition=N;
    int temp,i;

    for(i=2; i<=N; i++)
    {
        if(X[i]>v)
        {
            v=X[i];
            vposition=i;
        }
    }

    temp=X[N];
    X[N]=v;
    X[vposition]=temp;

    return v;
}

//Find the number of elements in each group
void set_Count(int u, int v)
```

```
{
   int i,j,jj;

   for(j=0; j<K; j++)
   {
      Count[j]=0;
   }


   for(i=1; i<=N; i++)
   {
      jj=(X[i]-u)/C;  //Calculate group number
      Count[jj]=Count[jj]+1;
   }
}

//Move elements to the first and the last group
void move_group(int u, int v)
{
   int m=2;
   int l=N-1;
   int i,j,jj,temp;

   for(i=2; i<N; i++)  //Move elements to the first group
   {
      if((j=(X[i]-u)/C)==0 && m<=Count[0])
      {
         temp=X[m];
         X[m]=X[i];
         X[i]=temp;
         m++;
      }
   }

   for(KK=K-1; KK>0; KK--)  //Find the last group
   {
      if(Count[KK]==0)
      {
         continue;
      }
      else
      {
         break;
      }
   }

   for(i=N-1; i>1; i--)  //Move elements to the last group
   {
      if((jj=(X[i]-u)/C)==KK && l>N-Count[KK])
      {
         temp=X[l];
         X[l]=X[i];
         X[i]=temp;
         l--;
      }
   }
}
```

```
//Move elements to their correct group
void group(int u, int v)
{
   int i,ii,j,jj,temp;
   int I;

   Count[KK]=N+1-Count[KK];  //Set start position of the last group

   for(j=KK-1; j>=0; j--)  //Set start position of each group
   {
      Count[j]=Count[j+1]-Count[j];
   }

   Count[0]=Count[1]-1;  //Get end position of the first group

   for(i=0; i<=KK; i++)  //Set non-element group position to 0
   {
      if(Count[i]==Count[i+1])
      {
         Count[i]=0;
      }
   }

   for(i=2; i<KK; i++)  //Set the end element of each group to negative
   {
      if(Count[i] != 0)
      {
         X[Count[i]-1]=0-X[Count[i]-1];
      }
   }

   //Pass through sorting array from the second group
   //to the last second group.
   i=Count[1];
   j=1;
   I=Count[KK-1];

   while(i<I && j<KK-1)
   {
      if(Count[j]>0)  //Group is not completed, continue move element
      {
         if(X[i]>0)  //Group is not completed, continue move element
         {
            jj=(X[i]-u)/C;
            if(j==jj)  //In correct group
            {
               if(i>=Count[j])
               {
                  Count[jj]++;
                  i++;
               }
               else
               {
                  i=Count[jj];
               }
            }
```

```
    else  //Not in correct group, exchange to correct group
    {
       if(X[Count[jj]]>0)  //Group is not completed
       {
          temp=X[Count[jj]];
          X[Count[jj]]=X[i];
          X[i]=temp;
          Count[jj]++;
       }
       else  //Group is completed
       {
          X[Count[jj]]=0-X[Count[jj]];
          temp=X[Count[jj]];
          X[Count[jj]]=0-X[i];
          X[i]=temp;
          Count[jj]=0-Count[jj];
       }
    }
  }

  else  //Group is completed
  {
     X[i]=0-X[i];
     jj=(X[i]-u)/C;
     if(j==jj)  //In correct group
     {
        i=Count[j]+1;
        Count[j]=0-Count[j];
        j++;
     }
     else  //Not in correct group
     {
        if(X[Count[jj]]>0)  //Group is not completed
        {
           temp=X[Count[jj]];
           X[Count[jj]]=X[i];
           X[i]=0-temp;
           Count[jj]++;
        }
        else  //Group is completed
        {
           X[Count[jj]]=0-X[Count[jj]];
           temp=X[Count[jj]];
           X[Count[jj]]=X[i];
           X[i]=0-temp;
           Count[jj]=0-Count[jj];
        }
     }
  }
}

else if(Count[j]<0)  //Group is completed, to the next group
{
   i=1-Count[j];
   j++;
}
```

```
      else if(Count[j]==0)  //Non-element group, to the next group
      {
         j++;
      }
   }

   //Get the end position of each group
   for(ii=0; ii<=KK; ii++)
   {
      Count[ii]=abs(Count[ii]);
   }
   if(Count[KK-1]==0)
   {
      Count[KK-1]=Count[KK]=N;
   }
   else
   {
      Count[KK-1]=Count[KK]-1;
      Count[KK]=N;
   }
   for(ii=KK-2; ii>=0; ii--)
   {
      if(Count[ii]==0)
      {
         Count[ii]=Count[ii+1];
      }
   }

   for(ii=0; ii<=N; ii++)  //Change all data to positive
   {
      X[ii]=abs(X[ii]);
   }

}

//Print the result to output file
void printout()
{
   int i;

   for(i=1; i<=N; i++)
   {
      if(!(i%10==0))
      {
         fprintf(op, "%d   ", X[i]);
      }

      else
      {
         fprintf(op, "%d   \n", X[i]);
      }
   }

   fprintf(op, "\n");

}
```

```
//Quicksort comparison function
int test(int *x, int *y)
{
   if(*x > *y)
      return (1);
   if(*x < *y)
      return (-1);
   return (0);
}
```

APPENDIX B

C PROGRAMMING CODE FOR FLASHSORT1

```
//Flashsort1:
//Sorted date type is integer with range 1 to 32767
//Read sorted data from input file and put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50000  //Maximum input array size

//Global variable
int A[MAX+1];  //Sorted data array
int L[MAX+1];  //Class vector array
int N;  //Number of sorted data
int M;  //Additional storage space
int ANMIN;  //Minimum value
int NMAX;  //Maximum value position
int ANMAX;  //Maximum value

//Input and output file pointer
FILE *ip;
FILE *op;

//Function declaration
void get_data(FILE *ip);
void scan_data(int *A);
double factor();
void classification();
void permutation();
void insertion(int class);
void printout();

//Main function
main(int argc, char *argv[])
{
   clock_t start_time, end_time;
   int class;

   if(argc != 3)
   {
      printf("Use 'a.out inputfile outputfile' to run\n");
      exit(0);
   }

   if((ip=fopen(argv[1], "r"))==NULL)  //Open input file
   {
      printf("Cannot open the input file! Try again!\n");
      exit(0);
   }

   if((op=fopen(argv[2], "w"))==NULL)  //Open output file
```

```
   {
      printf("Cannot open the output file! Try again!\n");
      exit(0);
   }

   get_data(ip);  //Read data from input file
   start_time=clock();  //Set starting time
   scan_data(A);  //Find the minimum and maximum value
   classification();  //Find the number of elements in each class
   permutation();  //Move elements to their correct class
   for(class=1; class<=M; class++)  //Sort each class
   {
      insertion(class);
   }
   end_time=clock();  //Get ending time
   printf("The run time is: %d\n", end_time-start_time);
   fprintf(op, "The run time is: %d\n", end_time-start_time);
   printout();  //Print the result to output file

   fclose(ip);
   fclose(op);

}

//Read data from input file
//Input file contains sorted data separated by blank space
void get_data(FILE *ip)
{
   int input;
   int count;
   int i;
   char temp[10];

   while(!feof(ip))
   {
      A[0]=0;

      for(count=1; fscanf(ip, "%d", &input)==1; count++)
      {
         if(count>MAX)
         {
            printf("Array is not enough to hold the elements to be
sorted!\n");
            exit(0);
         }

         A[count]=input; //Put data into sorted data array
      }
   }

   N=count-1;  //Number of sorted data
   printf("The sorting array size is: %d\n", N);
   printf("Please enter the additional storage space size: ");
   M=atoi(gets(temp));  //Size of additional storage space

   fprintf(op, "The sorting array size is: %d\n", N);
   fprintf(op, "The additional storage space is: %d\n", M);
```

```
    for(i=M; i>0; i--)  //Initial class vector
    {
       L[i]=0;
    }
    L[0]=0;

}

//Find the minimum and maximum value
void scan_data(int *A)
{
    int i;

    ANMIN=A[1];  //Initial minimum value
    NMAX=1;  //Initial maximum value position

    for(i=1; i<=N; i++)  //Scan data to find the minimum value
    {                    //and maximum value position
       if(A[i]<ANMIN)
          ANMIN=A[i];
       if(A[i]>A[NMAX])
          NMAX=i;
    }

    ANMAX=A[NMAX];  //Find maximum value
}

//Compute the constant factor
double factor()
{
    double C1;

    C1=((double)(M-1)/(ANMAX-ANMIN));

    return C1;
}

//Find the number of elements in each class
void classification()
{
    int i,k;
    int HOLD;
    double C1;

    if(ANMIN==ANMAX)
       return;

    for(k=1; k<=M; k++)
    {
       L[k]=0;
    }

    for(i=1; i<=N; i++)
    {
       C1=factor();
```

```
        k=1+(int)(C1*(A[i]-ANMIN));  //Calculate class number
        L[k]=L[k]+1;
    }

    for(k=1; k<=M; k++)  //Set end position of each class
    {
        L[k]=L[k]+L[k-1];
    }

    //Exchange the maximum value with the first data
    HOLD=ANMAX;
    A[NMAX]=A[1];
    A[1]=HOLD;

}

//Move elements to their correct class
void permutation()
{
    int i;
    int NMOVE=0;  //Number of elements being moved
    int J=1;  //Initial scan data position
    int k=M;  //Initial class number
    int HOLD;
    int FLASH;
    double C1;

    while(NMOVE<N)  //Scan all data
    {
        while(J>L[k])  //Class in completed
        {
            J=J+1;
            C1=factor();
            k=1+(int)(C1*(A[J]-ANMIN));
        }

        FLASH=A[J];  //Get next cycle to be sorted

        while(!(J==L[k]+1))  //Class in not completed
        {
            C1=factor();
            k=1+(int)(C1*(FLASH-ANMIN));
            HOLD=A[L[k]];
            A[L[k]]=FLASH;
            FLASH=HOLD;
            L[k]=L[k]-1;  //Complete an exchange
            NMOVE=NMOVE+1;  //Sort one more data
        }
    }

}

//Straight insertion to sort each class
void insertion(int class)
{
    int i,j,k,HOLD;
```

```
   for(i=L[class]+2; i<=L[class+1]; i++)
   {
         HOLD=A[i];
         j=i-1;

         while(j>0 && A[j]>HOLD)
         {
            A[j+1]=A[j];
            j=j-1;
         }

         A[j+1]=HOLD;
   }

}

//Print the result to output file
void printout()
{
   int i;

   for(i=1; i<=N; i++)
   {
      if(!(i%10==0))
      {
         fprintf(op, "%d   ", A[i]);
      }

      else
      {
         fprintf(op, "%d   \n", A[i]);
      }
   }

   fprintf(op, "\n");

}
```

APPENDIX C

C PROGRAMMING CODE FOR BUCKET SORT

```
//Bucket sort:
//Sorted date type is integer with range 1 to 32767
//Read sorted data from input file and put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50000  //Maximum input data array size

//Structure
struct node {
   int key;
   struct node *next;
};

typedef struct node LIST;

//Global variable
LIST *list=NULL;  //Sorted data list
LIST *local_list[MAX];  //Bucket array list
int N;  //Number of sorted data
int B;  //Bucket number
int K;  //Constant factor

//Input and output file pointer
FILE *ip;
FILE *op;

//Function declaration
void get_data(FILE *ip);
void printout(LIST *list);


//Main function
main(int argc, char *argv[])
{
   int i,j;
   clock_t start_time, end_time;
   LIST *tempp, *temppp, *temp1, *temp2, *temp3;

   if(argc != 3)
   {
      printf("Use 'a.out inputfile outputfile' to run\n");
      exit(0);
   }

   if((ip=fopen(argv[1], "r"))==NULL)  //Open input file
   {
      printf("Cannot open the input file! Try again!\n");
      exit(0);
```

```
}

if((op=fopen(argv[2], "w"))==NULL)  //Open output file
{
   printf("Cannot open the output file! Try again!\n");
   exit(0);
}

get_data(ip);  //Read data from input file
K=32767/B+1;
start_time=clock();  //Set starting time

tempp=(LIST *)malloc(sizeof(LIST));
tempp=list;

while(tempp != NULL)  //Scan all data and insert to correct bucket
{                     //with correct position
   j=tempp->key/K;  //Calculate bucket number
   temp1=(LIST *)malloc(sizeof(LIST));  //Make a node to be linked
   temp1->key=tempp->key;
   temp1->next=NULL;

   if(local_list[j]==NULL)  //First node in the bucket
   {
      local_list[j]=temp1;
   }

   else  //Not first node in the bucket
   {
      temp3=(LIST *)malloc(sizeof(LIST));
      temp3=local_list[j];
      if(temp3->next == NULL)  //Only one node in the bucket
      {
         if(temp3->key <= temp1->key)  //Insert at the end
         {
            temp3->next=temp1;
         }
         else  //Insert at the beginning
         {
            temp1->next=temp3;
            local_list[j]=temp1;
         }
      }
      else{  //More than one node in the bucket
      while(temp3 != NULL)  //Find the correct position to insert
      {
         if(temp3->key<=temp1->key && temp3->next==NULL)
         {  //Insert at the end
            temp3->next=temp1;
            break;  //Go to next data
         }
         else if(temp3->key<=temp1->key && temp3->next->key>temp1-
                 >key && temp3->next != NULL)
         {  //Insert between two node
            temp1->next=temp3->next;
            temp3->next=temp1;
            break;  //Go to next data
```

```
            }
            else if(temp3->key<=temp1->key && temp3->next->key<=temp1-
                    >key && temp3->next != NULL)
            {  //Not find the insert position
               temp3=temp3->next;
            }
            else if(temp3->key > temp1->key)
            {  //Insert at the front
               temp1->next=temp3;
               local_list[j]=temp1;
               break;  //Go to next data
            }
         }}
      }

      tempp=tempp->next;
   }

   //Link the sorted data to list
   list=local_list[0];
   for(j=1; j<=B; j++)
   {
      while(local_list[0]->next != NULL)
      {
         local_list[0]=local_list[0]->next;
      }
      local_list[0]->next=local_list[j];
   }

   end_time=clock();  //Get ending time
   printf("The running time is: %d\n", end_time-start_time);
   fprintf(op, "The running time is: %d\n", end_time-start_time);
   printout(list);  //Print the result to output file

   fclose(ip);
   fclose(op);

}

//Read data from input file
//Input file contains sorted data separated by blank space
void get_data(FILE *ip)
{
   int input;
   int count;
   int i;
   char temp[10];
   LIST *new;

   while(!feof(ip)) //Put data into sorted data list
   {
      for(count=0; fscanf(ip, "%d", &input)==1; count++)
      {
         if(count>MAX)
         {
            printf("Array is not enough to hold the elements to be
sorted!\n");
```

```
            exit(0);
        }
        new=(LIST *)malloc(sizeof(LIST));
        new->key=input;
        new->next=NULL;
        if(list==NULL)
        {
            list=new;
        }
        else
        {
            new->next=list;
            list=new;
        }
    }
}


N=count;  //Number of sorted data
printf("The sorting array size is: %d\n", N);
printf("Please enter the bucket number: ");
B=atoi(gets(temp));  //Bucket number
for(i=0; i<B; i++)  //Initial each bucket array list
{
    local_list[i]=NULL;
}

fprintf(op, "The sorting array size is: %d\n", N);
fprintf(op, "The bucket number is: %d\n", B);

}

//Print the result to output file
void printout(LIST *list)
{
    int i;
    LIST *ttemp;

    ttemp=(LIST *)malloc(sizeof(LIST));
    ttemp=list;

    for(i=1; i<=N; i++)
    {
        if(!(i%10==0))
        {
            fprintf(op, "%d    ", ttemp->key);
            ttemp=ttemp->next;
        }
        else
        {
            fprintf(op, "%d    \n", ttemp->key);
            ttemp=ttemp->next;
        }
    }

    fprintf(op, "\n");
}
```

APPENDIX D

C PROGRAMMING CODE FOR QUICKSORT

```
//Quicksort:
//Sorted date type is integer with range 1 to 32767
//Read sorted data from input file and put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50000  //Maximum input array size

int X[MAX];  //Sorted data array
int N;  //Number of sorted data

//Input and output file pointer
FILE *ip;
FILE *op;

//Function declaration
void get_data(FILE *ip);
void printout();
int test(int *x, int *y);


//Main function
main(int argc, char *argv[])
{
   int i;
   clock_t start_time, end_time;

   if(argc != 3)
   {
      printf("Use 'a.out inputfile outputfile' to run\n");
      exit(0);
   }

   if((ip=fopen(argv[1], "r"))==NULL)  //Open input file
   {
      printf("Cannot open the input file! Try again!\n");
      exit(0);
   }

   if((op=fopen(argv[2], "w"))==NULL)  //Open output file
   {
      printf("Cannot open the output file! Try again!\n");
      exit(0);
   }

   get_data(ip);  //Read data from input file
   start_time=clock();  //Set starting time
   qsort(&X[1],N,sizeof(X[1]),test);  //Quicksort
   end_time=clock();  //Get ending time
```

```
   printf("The run time is: %d\n", end_time-start_time);
   fprintf(op, "The run time is: %d\n", end_time-start_time);
   printout();  //Print the result to output file

   fclose(ip);
   fclose(op);
}

//Read data from input file
//Input file contains sorted data separated by blank space
void get_data(FILE *ip)
{
   int input;
   int count;
   int i;
   char temp[10];

   while(!feof(ip))
   {
      X[0]=0;

      for(count=1; fscanf(ip, "%d", &input)==1; count++)
      {
         if(count>MAX)
         {
            printf("Array is not enough to hold the elements to be
sorted!\n");
            exit(0);
         }

         X[count]=input;  //Put data into sorted data array
      }
   }

   N=count-1;  //Number of sorted data
   printf("The sorting array size is: %d\n", N);

   fprintf(op, "The sorting array size is: %d\n", N);
}

//Quicksort comparison function
int test(int *x, int *y)
{
   if(*x > *y)
      return (1);
   if(*x < *y)
      return (-1);
   return (0);
}

//Print the result to output file
void printout()
{
   int i;

   for(i=1; i<=N; i++)
   {
```

```
        if(!(i%10==0))
        {
            fprintf(op, "%d    ", X[i]);
        }

        else
        {
            fprintf(op, "%d    \n", X[i]);
        }
    }

    fprintf(op, "\n");

}
```

APPENDIX E

C PROGRAMMING CODE FOR GENERATING UNIFORMLY DISTRIBUTED INPUT ARRAY

```
//Generate unifomly distributed integers
//Integers are with range 1 to 32767
//Put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 50000  //Maximum array size
//Main function
main(int argc, char *argv[])
{
   unsigned int seed;  //Seed used to generate next random number
   int count[MAX];  //Count array
   int hold[MAX];  //Random number array
   int array[MAX];  //Sorted data array
   int i,j;
   int N;  //Number of sorted data
   int M;  //Size of additional storage space
   char temp[10],temp1[10];

   FILE *op;  //Output file pointer

   if(argc != 2)
   {
      printf("Use 'a.out outputfile' to run\n");
      exit(0);
   }

   if((op=fopen(argv[1], "w"))==NULL)  //Open output file
   {
      printf("Cannot open the output file! Try again!\n");
      exit(0);
   }

   printf("Please enter sorting array size: ");
   N=atoi(gets(temp));  //Number of sorted data
   printf("Please enter additional storage space size: ");
   M=atoi(gets(temp1));  //Size of additional storage space

   seed=(unsigned int)time(NULL);  //Get seed

   srand(seed);  //Seed random number

   for(i=0; i<MAX; i++)  //Initial count array to 0
   {
      count[i]=0;
   }

   for(i=0,j=0; i<MAX && j<N; i++)
```

```
{   //Generate uniform integers with range from 1 to 32767
    hold[i]=rand()%32767;   //Get random number
    if(count[hold[i]/(32767/M)]<(N/M) && hold[i] != 0)
    {   //Take uniform integer into sorted data array
        count[hold[i]/(32767/M)]++;
        array[j++]=hold[i];
    }
}

for(i=0; i<j; i++)   //Print sorted data to output file
{
    if(!(i%10==0))
    {
        fprintf(op, "%d    ", array[i]);
    }
    else
    {
        fprintf(op, "%d    \n", array[i]);
    }
}
fprintf(op, "\n");

fclose(op);
}
```

APPENDIX F

C PROGRAMMING CODE FOR GENERATING TRUNCATED NORMAL DISTRIBUTED INPUT ARRAY

```
//Generate normal random integers
//Integers are at range 1 to 32767 with parameter u = 16384
//Put result to output file
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define MAX 50000  //Maximum input array size


//Main function
main(int argc, char *argv[])
{
   unsigned int seed;  //Seed used to generate next random number
   int count[MAX];  //Count array
   int array[MAX];  //Sorted data array
   int i,j;
   int N;  //Number of sorted data
   int D;  //Stand deviation
   char temp[10],temp1[10];
   int u=16384;  //Parameter used to generate normal random integer
   int X;  //Random integer with range 1 to 32767
   double p;
   double f;  //Density of X
   double range;  //Count number of X
   double a,b,c,d;

   FILE *op;  //Output file pointer

   if(argc != 2)
   {
      printf("Use 'a.out outputfile' to run\n");
      exit(0);
   }

   if((op=fopen(argv[1], "w"))==NULL)  //Open output file
   {
      printf("Cannot open the output file! Try again!\n");
      exit(0);
   }

   printf("Please enter sorting array size: ");
   N=atoi(gets(temp));  //Number of sorted data
   printf("Please enter stand deviation: ");
   D=atoi(gets(temp1));  //Stand deviation

   seed=(unsigned int)time(NULL);  //Get seed
```

```
srand(seed);  //Seed random number

for(i=0; i<MAX; i++)  //Initial count array to 0
{
   count[i]=0;
}

for(j=1; j<N-1; )
    {  // Generate normal random integers with range from 1 to 32767
   X=rand()%32767;  //Get random integer
   a=pow((double)(X-u),2.0);
   b=pow((double)D,2.0);
   p=(0.0-a)/(2.0*b);
   c=pow(M_E,p);
   d=sqrt(M_PI);
   f=c/(M_SQRT2*d*(double)D);  //Calculate density of X
   range=f*(double)N;  //Get count number of X
   if((double)count[X]<range && X != 0)
            {  //Take normal random integer into sorted data array
      count[X]++;
      array[j++]=X;
    }
 }
 array[0]=1;
 array[N-1]=32767;

 for(i=0; i<N; i++)  //Print sorted data to output file
 {
    if(!(i%10==0))
    {
       fprintf(op, "%d   ", array[i]);
    }
    else
    {
       fprintf(op, "%d   \n", array[i]);
    }
 }
 fprintf(op, "\n");

 fclose(op);
}
```

VITA

Zhimin Ma

Candidate for the Degree of

Master of Science

Thesis: STUDY OF NON-COMPARISON SORTING ALGORITHMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Baotou, Nei Mongol Province of P.R.China, the daughter of Qinggui Ma and Chenggui Zhang.

Education: Graduate from the No. 1 middle school of Baogang in July, 1986; received Bachelor of Science degree in Material Science and Engineering from Beijing University of Aeronautics & Astronauntics in July, 1990. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 2000.

Experience: Worked as an engineer in Beijing Research Institute of Material & Technology from 1990 to 1996; employed as a graduate research assistant in Chemistry Department of Oklahoma State University from January, 1999 to December, 1999.